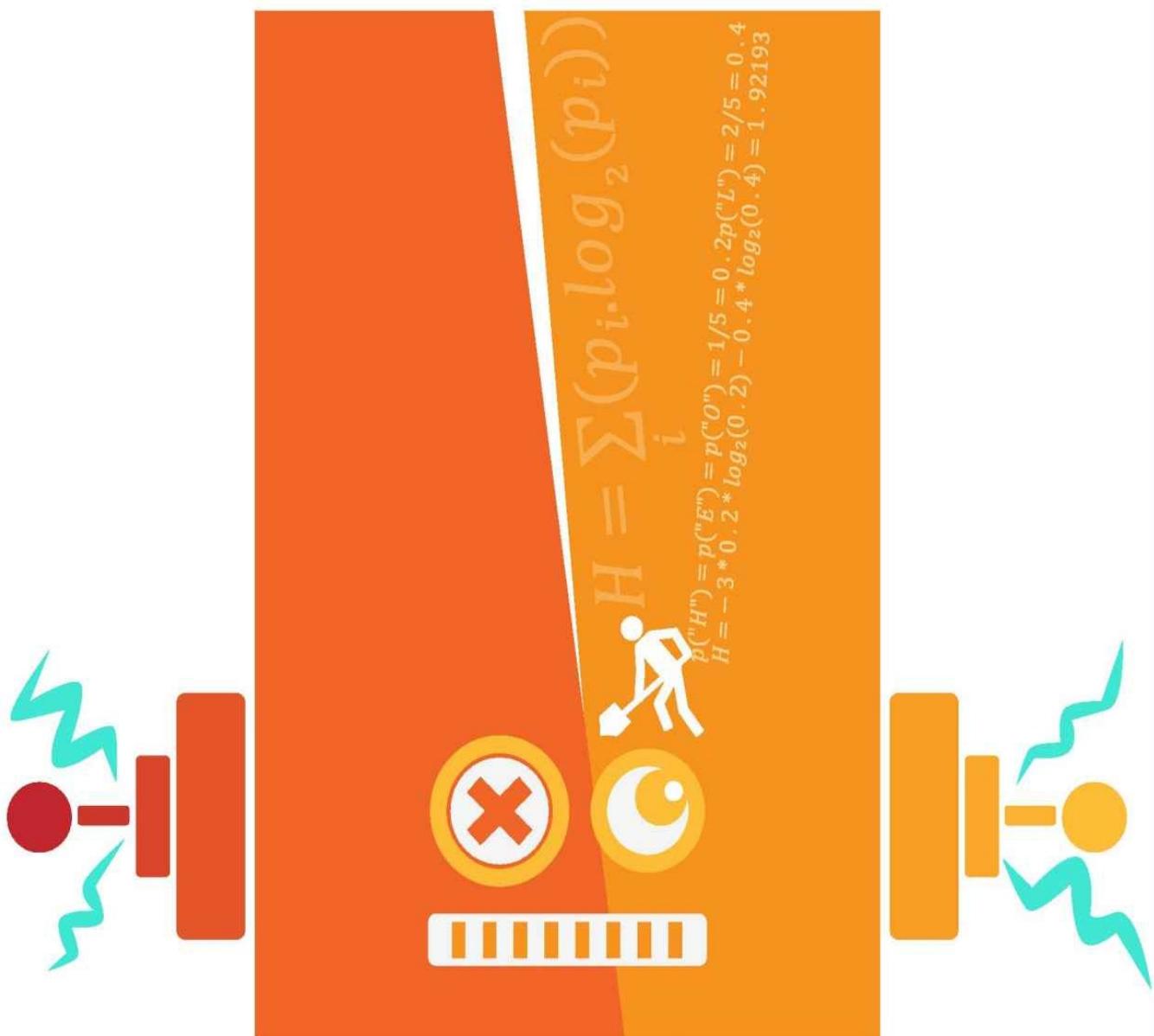


BLEEDING EDGE PRESS

TENSORFLOW FOR MACHINE INTELLIGENCE



**Sam Abrahams, Danijar Hafner,
Erik Erwitt, Ariel Scarpinelli**

TensorFlow for Machine Intelligence

A Hands-On Introduction to Learning Algorithms

Sam Abrahams

Danijar Hafner

Erik Erwitt

Ariel Scarpinelli

TensorFlow for Machine Intelligence

Copyright (c) 2016 Bleeding Edge Press

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

This book expresses the authors views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Bleeding Edge Press, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

ISBN: 978-1-939902-35-1

Published by: Bleeding Edge Press, Santa Rosa, CA 95404

Title: TensorFlow for Machine Intelligence

Authors: Sam Abrahams, Danijar Hafner, Erik Erwitt, Ariel Scarpinelli

Acquisitions Editor: Christina Rudloff

Editor: Troy Mott

Cover Designer: Martin Murtonen

Website: bleedingedgepress.com

Preface

Welcome

Since its open source release in November 2015, [TensorFlow](#) has become one of the most exciting machine learning libraries available. It is being used more and more in research, production, and education. The library has seen continual improvements, additions, and optimizations, and the TensorFlow community has grown dramatically. With [TensorFlow for Machine Intelligence](#), we hope to help new and experienced users hone their abilities with TensorFlow and become fluent in using this powerful library to its fullest!

Background education

While this book is primarily focused on the TensorFlow API, we expect you to have familiarity with a number of mathematical and programmatic concepts. These include:

- Derivative calculus (single-variable and multi-variables)
- Matrix algebra (matrix multiplication especially)
- Basic understanding of programming principles
- Basic machine learning concepts

In addition to the above, you will get more out of this book if they have the following knowledge:

- Experience with Python and organizing modules
- Experience with the [NumPy](#) library
- Experience with the [matplotlib](#) library
- Knowledge of more advanced machine learning concepts, especially feed-forward neural networks, convolutional neural networks, and recurrent neural networks

When appropriate, we'll include refresher information to re-familiarize the reader with some of the concepts they need to know, to fully understand the math and/or Python concepts.

What you should expect to learn

This TensorFlow book introduces the framework and the underlying machine learning concepts that are important to harness machine intelligence.

After reading this book, you should know the following:

- A deep understanding of the core TensorFlow API
- The TensorFlow workflow: graph definition and graph execution
- How to install TensorFlow on various devices
- Best practices for structuring your code and project
- How to create core machine learning models in TensorFlow
- How to implement RNNs and CNNs in TensorFlow
- How to deploy code with TensorFlow Serving
- The fundamentals of utilizing TensorBoard to analyze your models

This book's layout

Section 1: Getting started with TensorFlow

The first section of this book helps get you on your feet and ready to use TensorFlow. The first chapter is the introduction, which provides a brief historical context for TensorFlow and includes a discussion about TensorFlow's design patterns as well as the merits and challenges of choosing TensorFlow as a deep learning library.

After the introduction, “TensorFlow Installation” goes over some considerations a person installing TensorFlow should think about, and includes detailed instructions on installing TensorFlow: both installing from binaries as well as building from source.

Section 2: TensorFlow and Machine Learning fundamentals

This second section begins in “Fundamentals of TensorFlow” by getting TensorFlow installed on your machine and diving deep into the fundamentals of the TensorFlow API *without* incorporating a lot of machine learning concepts. The goal is to isolate “learning TensorFlow” and “learning how to do machine learning with TensorFlow”. This will include an in depth description of many of the most important pieces of the TensorFlow API. We’ll also show how you can take a visual graph representation of a model and translate it into TensorFlow code, as well as verify that the graph is modeled correctly by using TensorBoard.

Once the core API concepts are covered, we continue with “Machine Learning Basics”, in which we create simple machine learning models, such as linear regression, logistic regression, and clustering, inside of TensorFlow.

Section 3: Implementing advanced deep models in TensorFlow

The third section is comprised of two chapters, each focusing on a different type of more complex deep learning model. Each will describe the model in question and present visual graph representation of what we are trying to create. We'll discuss why the model is setup the way it is, note any mathematical quirks, and then go over how to set them up effectively in TensorFlow.

The first model we'll look at is the Convolutional Neural Network, or CNN, in "Object Recognition and Classification," where we'll talk about training TensorFlow models on image data. This will include a discussion on the math and purpose of convolutions, how to convert raw image data into a TensorFlow-compatible format, and how to test your final output.

In the chapter "Natural Language Processing with Recurrent Networks," we'll examine how to properly create Recurrent Neural Networks, or RNNs, in TensorFlow. Looking at a variety of natural language processing (NLP) tasks, we'll see how to use Long Short-Term Memory (networks) and incorporate pre-trained word vectors in your model.

Section 4: Additional tips, techniques, and features

The final section of the book will explore the latest features available in the TensorFlow API. Topics include preparing your model for deployment, useful programming patterns, and other select subjects.

Other machine learning libraries

TensorFlow is not the only open source machine learning library out there. Below is a short list of various options for deep learning:

- [Caffe](#) focuses on convolutional neural networks and image processing, and is written in C++.
- [Chainer](#) is another flexible machine learning Python library capable of utilizing multiple GPUs on one machine.
- [CNTK](#) is Microsoft's entry into the open source machine learning library game. It uses its own model definition language to build distributed models declaratively.
- [Deeplearning4j](#) is a Java library specifically focused on neural networks. It is built to be scaled and integrated with Spark, Hadoop, and other Java-based distributed software.
- [Nervana Neon](#) is an efficient Python machine learning library, which is capable of using multiple GPUs on a single machine.
- [Theano](#) is an extremely flexible Python machine learning library written in Python. It is popular in research, as it is quite user friendly and capable of defining complex models fairly easily. TensorFlow's API is most similar to Theano's.
- [Torch](#) is a machine learning library that focuses on GPU implementation. It is written in Lua, and is backed by research teams at several large companies.

It's beyond the scope of this book to have an in-depth discussion on the merits of each of these libraries, but it is worth looking into them if you have the time. The authors of TensorFlow took inspiration from several of them when designing the framework.

Further reading

If, after reading this book, you're interested in pursuing more with TensorFlow, here are a couple of valuable resources:

- [The official TensorFlow website](#), which will contain the latest documentation, API, and tutorials
- [The TensorFlow GitHub repository](#), where you can contribute to the open-source implementation of TensorFlow, as well as review the source code directly
- [Officially released machine learning models implemented in TensorFlow](#). These models can be used as-is or be tweaked to suit your own goals
- [The Google Research Blog](#) provides the latest news from Google related to TensorFlow applications and updates.
- [Kaggle](#) is a wonderful place to find public datasets and compete with other data-minded people
- [Data.gov](#) is the U.S. government's portal to find public datasets all across the United States

Alright, that's enough of a pep-talk. Let's get started with [TensorFlow for Machine Intelligence!](#)

Part I. Getting started with TensorFlow

Chapter 1. Introduction

Data is everywhere

We are truly in “The Information Age.” These days, data flows in from everywhere: smartphones, watches, vehicles, parking meters, household appliances- almost any piece of technology you can name is now being built to communicate back to a database somewhere in the cloud. With access to seemingly unlimited storage capacity, developers have opted for a “more-is-better” approach to data warehousing, housing petabytes of data gleaned from their products and customers.

At the same time, computational capabilities continue to climb. While the growth of CPU speeds has slowed, there has been an explosion of parallel processing architectures. Graphics processing units (GPUs), once used primarily for computer games are now being used for general purpose computing, and they have opened the floodgates for the rise of machine learning.

Machine learning, sometimes abbreviated to “ML,” uses general-purpose mathematical models to answer specific questions using data. Machine learning has been used to detect spam email, recommend products to customers, and predict the value of commodities for many years. In recent years, a particular kind of machine learning has seen an incredible amount of success across all fields: deep learning.

Deep learning

“Deep learning” has become the term used to describe the process of using multi-layer neural networks- incredibly flexible models that can use a huge variety and combination of different mathematical techniques. They are incredibly powerful, but our ability to utilize neural networks to great effect has been a relatively new phenomena, as we have only recently hit the critical mass of data availability and computational power necessary to boost their capabilities beyond those of other ML techniques.

The power of deep learning is that it gives the model more flexibility in deciding how to use data to best effect. Instead of a person having to make wild guesses as to which inputs are worth including, a properly tuned deep learning model can take all parameters and automatically determine useful, higher-order combinations of its input values. This enables a much more sophisticated decision-making process, making computers more intelligent than ever. With deep learning, we are capable of creating cars that drive themselves and phones that understand our speech. Machine translation, facial recognition, predictive analytics, machine music composition, and countless artificial intelligence tasks have become possible or significantly improved due to deep learning.

While the mathematical concepts behind deep learning have been around for decades, programming libraries dedicated to creating and training these deep models have only been available in recent years. Unfortunately, most of these libraries have a large trade-off between flexibility and production-worthiness. Flexible libraries are invaluable for researching novel model architectures, but are often either too slow or incapable of being used in production. On the other hand, fast, efficient, libraries which can be hosted on distributed hardware are available, but they often specialize in specific types of neural networks and aren't suited to researching new and better models. This leaves decision makers with a dilemma: should we attempt to do research with inflexible libraries so that we don't have to reimplement code, or should we use one library for research and a completely different library for production? If we choose the former, we may be unable to test out different types of neural network models; if we choose the latter, we have to maintain code that may have completely different APIs. Do we even have the resources for this?

TensorFlow aims to solve this dilemma.

TensorFlow: a modern machine learning library

TensorFlow, open sourced to the public by Google in November 2015, is the result of years of lessons learned from creating and using its predecessor, DistBelief. It was made to be flexible, efficient, extensible, and portable. Computers of any shape and size can run it, from smartphones all the way up to huge computing clusters. It comes with lightweight software that can instantly productionize your trained model, effectively eliminating the need to reimplement models. TensorFlow embraces the innovation and community-engagement of open source, but has the support, guidance, and stability of a large corporation. Because of its multitude of strengths, TensorFlow is appropriate for individuals and businesses ranging from startups to companies as large as, well, Google.

If you and your colleagues have data, a question to answer, and a computer that turns on, you're in luck- as TensorFlow could be the missing piece you've been looking for.

TensorFlow: a technical overview

This section aims to provide high level information about the TensorFlow library, such as what it is, its development history, use cases, and how it stacks up against competitors. Decision makers, stakeholders, and anyone who wants to understand the background of TensorFlow will benefit from reading this section.

A brief history of deep learning at Google

Google's original large-scale deep learning tool was DistBelief, a product of the Google Brain team. Since its creation, it has been used by dozens of teams for countless projects involving deep neural networks. However, as with many first-of-its-kind engineering projects, there were design mistakes that have limited the usability and flexibility of DistBelief. Sometime after the creation of DistBelief, Google began working on its successor, whose design would apply lessons learned from the usage and limitations of the original DistBelief. This project became TensorFlow, which was released to the public in November 2015. It quickly turned into a popular library for machine learning, and it is currently being used for natural language processing, artificial intelligence, computer vision, and predictive analytics.

What is TensorFlow?

Let's take a high-level view of TensorFlow to get an understanding what problems it is trying to solve.

Breaking down the one-sentence description

Looking at the [TensorFlow website](#), the very first words greeting visitors is the following (rather vague) proclamation:

TensorFlow is an open source software library for machine intelligence.

Just below, in the first paragraph under “About TensorFlow,” we are given an alternative description:

TensorFlow™ is an open source software library for numerical computation using data flow graphs.

This second definition is a bit more specific, but may not be the most comprehensible explanation for those with less mathematical or technical backgrounds. Let’s break it down into chunks and figure out what each piece means.

Open source:

TensorFlow was originally created by Google as an internal machine learning tool, but an implementation of it was open sourced under the [Apache 2.0 License](#) in November 2015. As open source software, anyone is allowed to download, modify, and use the code. Open source engineers can make additions/improvements to the code and propose their changes to be included in a future release. Due to the popularity TensorFlow has gained, there are improvements being made to the library on a daily basis- created by both Google and third-party developers.

Notice that we say “an implementation” and not “TensorFlow” was open sourced. Technically speaking, TensorFlow is an interface for numerical computation as described in the [TensorFlow white paper](#), and Google still maintains its own internal implementation of it. However, the differences between the open source implementation and Google’s internal implementation are due to connections to other internal software, and **not** Google “hoarding the good stuff”. Google is constantly pushing internal improvements to the public repository, and for all intents and purposes the open source release contains the same capabilities as Google’s internal version.

For the rest of this book, when we say “TensorFlow”, we are referring to the open source implementation.

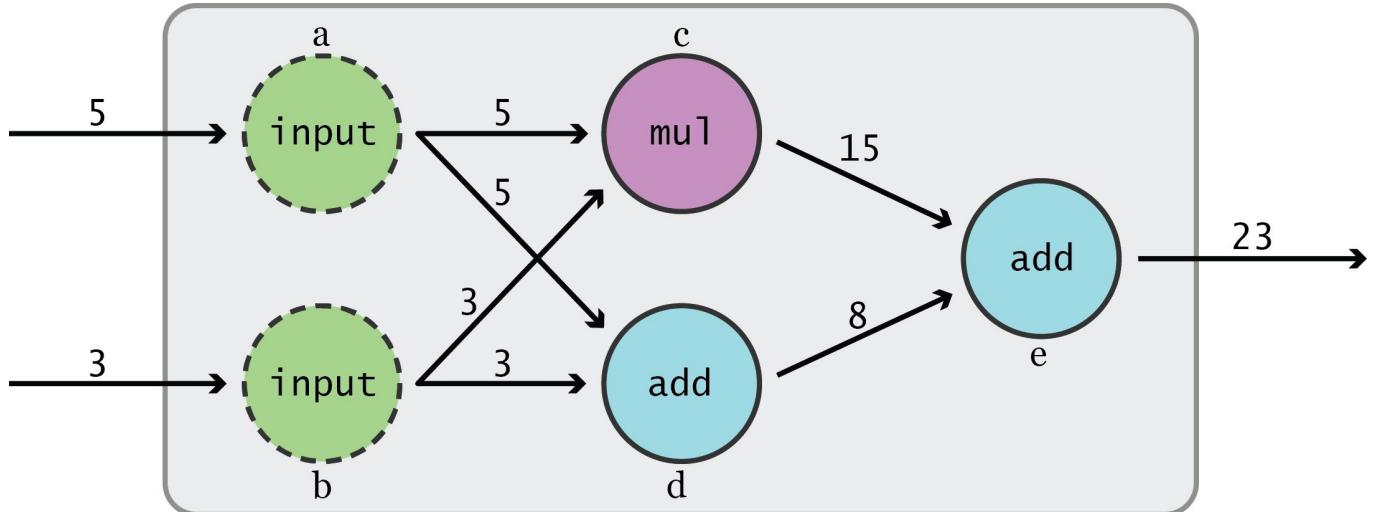
Library for numerical computation

Instead of calling itself a “library for machine learning”, it uses the broader term “numerical computation.” While TensorFlow does contain a package, [“learn” \(AKA “Scikit Flow”\)](#), that emulates the one-line modeling functionality of [Scikit-Learn](#), it’s important to note that TensorFlow’s primary purpose is *not* to provide out-of-the-box machine learning solutions. Instead, TensorFlow provides an extensive suite of functions and classes that allow users to define models from scratch mathematically. This allows users with the appropriate technical background to create customized, flexible models quickly and intuitively. Additionally, while TensorFlow does have extensive support for ML-specific functionality, it is just as well suited to performing complex mathematical computations. However, since this book is focused on machine learning (and deep

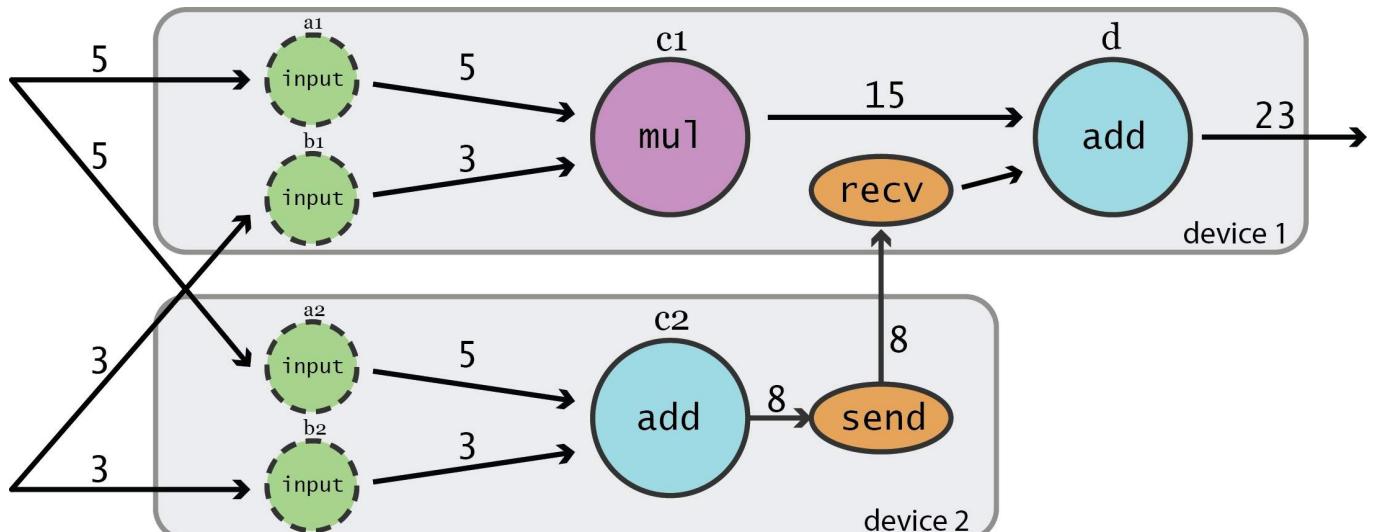
learning in particular), we will usually talk about TensorFlow being used to create machine learning models.

Data flow graphs

The computational model for TensorFlow is a [directed graph](#), where the *nodes* (typically represented by circles or boxes) are functions/computations, and the *edges* (typically represented by arrows or lines) are numbers, matrices, or tensors.



There are a number of reasons this is useful. First, many common machine learning models, such as [neural networks](#), are commonly taught and visualized as directed graphs already, which makes their implementation more natural for machine learning practitioners. Second, by splitting up computation into small, easily differentiable pieces, TensorFlow is able to automatically compute the derivative of any node (or “Operation”, as they’re called in TensorFlow) with respect to any other node that can affect the first node’s output. Being able to compute the derivative/gradient of nodes, especially output nodes, is crucial for setting up machine learning models. Finally, by having the computation separated, it makes it much easier to distribute work across multiple CPUs, GPUs, and other computational devices. Simply split the whole, larger graph into several smaller graphs and give each device a separate part of the graph to work on (with a touch of logic to coordinate sharing information across devices)



Quick aside: what is a tensor?

A tensor, put simply, is an n-dimensional matrix. So a 2-dimensional tensor is the same as a standard matrix. Visually, if we view an $m \times m$ matrix as a square array of numbers (m numbers tall, and m numbers wide), we can view an $m \times m \times m$ tensor as a cube array of numbers (m numbers tall, m numbers wide, and m numbers deep). In general, you can think about tensors the same way you would matrices, if you are more comfortable with matrix math!

Beyond the one-sentence description

The phrase “open source software library for numerical computation using data flow graphs” is an impressive feat of information density, but it misses several important aspects of TensorFlow that make it stand out as a machine learning library. Here are a few more components that also help make TensorFlow what it is:

Distributed

As alluded to when described data flow graphs above, TensorFlow is designed to be scalable across multiple computers, as well as multiple CPUs and GPUs within single machines. Although the original open source implementation did *not* have distributed capabilities upon release, as of version 0.8.0 the distributed runtime is available as part of the TensorFlow built-in library. While this initial distributed API is a bit cumbersome, it is incredibly powerful. Most other machine learning libraries do not have such capabilities, and it’s important to note that native compatibility with certain cluster managers (such as [Kubernetes](#)) are being worked on.

A suite of software

While “TensorFlow” is primarily used to refer to the API used to build and train machine learning models, TensorFlow is really a bundle of software designed to be used in tandem with:

- **[TensorFlow](#)** is the API for defining machine learning models, training them with data, and exporting them for further use. The primary API is accessed through Python, while the actual computation is written in C++ . This enables data scientists and engineers to utilize a more user-friendly environment in Python, while the actual computation is done with fast, compiled C++ code. There is a C++ API for executing TensorFlow models, but it is limited at this time and not recommended for most users.
- **[TensorBoard](#)** is graph visualization software that is included with any standard TensorFlow installation. When a user includes certain TensorBoard-specific operations in TensorFlow, TensorBoard is able to read the files exported by a TensorFlow graph and can give insight into a model’s behavior. It’s useful for summary statistics, analyzing training, and debugging your TensorFlow code. Learning to use TensorBoard early and often will make working with TensorFlow that much more enjoyable and productive.
- **[TensorFlow Serving](#)** is software that facilitates easy deployment of pre-trained TensorFlow models. Using built-in TensorFlow functions, a user can export their model to a file which can then be read natively by TensorFlow Serving. It is then able to start a simple, high-performance server that can take input data, pass it to the trained model, and return the output from the model. Additionally, TensorFlow Serving is capable of seamlessly switching out old models with new ones, without any downtime for end-users. While Serving is possibly the least recognized portion

of the TensorFlow ecosystem, it may be what sets TensorFlow apart from its competition. Incorporating Serving into a production environment enables users to avoid reimplementing their model, who can instead just pass along their TensorFlow export. TensorFlow Serving is written entirely in C++ , and its API is only accessible through C++.

We believe that using TensorFlow to its fullest means knowing how to use all of the above in conjunction with one another. Hence, we will be covering all three pieces of software in this book.

When to use TensorFlow

Let's take a look at some use cases for TensorFlow. In general, TensorFlow is generally a good choice for most machine learning purposes. Below is a short list of uses that TensorFlow is specifically targeted at.

Researching, developing, and iterating through new machine learning architectures. Because TensorFlow is incredibly flexible, it's useful when creating novel, less-tested models. With some libraries, you are given rigid, pre-built models that are good for prototyping, but are incapable of being modified.

Taking models directly from training to deployment. As described earlier, TensorFlow Serving enables you to quickly move from training to deployment. As such, it allows for a much faster iteration when creating a product that depends on having a model up-and-running. If your team needs to move fast, or if you simply don't have the resources to reimplement a model in C++, Java, etc., TensorFlow can give your team the ability to get your product off the ground.

Implementing existing complex architectures. Once you learn how to look at a visualization of a graph and build it in TensorFlow, you will be able to take models from recent research literature and implement them in TensorFlow. Doing so can provide insight when building future models or even give a strict improvement over the user's current model.

Large-scale distributed models. TensorFlow is incredibly good at scaling up over many devices, and it's already begun to replace DistBelief for various projects within Google. With the recent release of the distributed runtime, we'll see more and more cases of TensorFlow being run on multiple hardware servers as well as many virtual machines in the cloud.

Create and train models for mobile/embedded systems. While much of the attention on TensorFlow has been about its ability to scale up, it is also more than capable of scaling *down*. The flexibility of TensorFlow extends to systems with less computation power, and can be run on Android devices as well as mini-computers such as the [Raspberry Pi](#). The TensorFlow repository includes [an example on running a pre-trained model on Android](#).

TensorFlow's strengths

Usability

- The TensorFlow workflow is relatively easy to wrap your head around, and its consistent API means that you don't need to learn an entire new way to work when you try out different models.
- TensorFlow's API is stable, and the maintainers fight to ensure that every incorporated change is backwards-compatible.
- TensorFlow integrates seamlessly with Numpy, which will make most Python-savvy data scientists feel right at home.
- Unlike some other libraries, TensorFlow does not have any compile time. This allows you to iterate more quickly over ideas without sitting around.
- There are multiple higher-level interfaces built on top of TensorFlow already, such as [Keras](#) and [SkFlow](#). This makes it possible to use the benefits of TensorFlow even if a user doesn't want to implement the entire model by hand.

Flexibility

- TensorFlow is capable of running on machines of all shapes and sizes. This allows it to be useful from supercomputers all the way down to embedded systems- and everything in between.
- Its distributed architecture allows it to train models with massive datasets in a reasonable amount of time.
- TensorFlow can utilize CPUs, GPUs, or both at the same time.

Efficiency

- When TensorFlow was first released, it was surprisingly slow on a number of popular machine learning benchmarks. Since that time, the development team has devoted a ton of time and effort into improving the implementation of much of TensorFlow's code. The result is that TensorFlow now boasts impressive times for much of its library, vying for the top spot amongst the open-source machine learning frameworks.
- TensorFlow's efficiency is still improving as more and more developers work towards better implementations.

Support

- TensorFlow is backed by Google. Google is throwing a *ton* of resources into TensorFlow, since it wants TensorFlow to be the *lingua franca* of machine learning researchers and developers. Additionally, Google uses TensorFlow in its own work daily, and is invested in the continued support of TensorFlow.
- An incredible community has developed around TensorFlow, and it's relatively easy to get responses from informed members of the the community or developers on GitHub.

- Google has released several [pre-trained machine learning models in TensorFlow](#). They are free to use and can allow prototypes to get off the ground without needing massive data pipelines.

Extra features

- TensorBoard is invaluable when debugging and visualizing your model, and there is nothing quite like it available in other machine learning libraries.
- TensorFlow Serving may be the piece of software that allows more startup companies to devote services and resources to machine learning, as the cost of reimplementing code in order to deploy a model is no joke.

Challenges when using TensorFlow

Distributed support is still maturing

Although it is officially released, using the distributed features in TensorFlow is not as smooth as you might expect. As of this writing, it requires manually defining the role of each device, which is tedious and error-prone. Because it is brand new, there are less examples to learn from, although this should improve in the future. As mentioned earlier, support for Kubernetes is currently in the development pipeline, but for now it's still a work in progress.

Implementing custom code is tricky

There is an [official how-to on creating your own operations in TensorFlow](#), but there is a fair amount of overhead involved when implementing customized code into TensorFlow. If, however, you are hoping to contribute it to the master repository, the Google development team is quick to help answer questions and look over your code to bring your work into the fold.

Certain features are still missing

If you are an experienced machine learning professional with a ton of knowledge about a different framework, it's likely that there is going to be a small but useful feature you like that hasn't been implemented in TensorFlow yet. Often, there is a way to get around it, but that won't stop you from saying "Why isn't this natively supported yet?"

Onwards and upwards!

Needless to say, we are incredibly excited about the future of TensorFlow, and we are thrilled to give you a running start with such a powerful tool. In the next chapter, you'll install TensorFlow and learn all about the core TensorFlow library, basic use patterns, and environment.

Chapter 2. TensorFlow Installation

Before you get started using TensorFlow, you'll need to install the software onto your machine. Fortunately, the official TensorFlow website provides a [complete, step-by-step guide to installing TensorFlow](#) onto Linux and Mac OS X computers. This chapter provides our recommendations for different options available for your installation, as well as information regarding additional third-party software that integrates well with TensorFlow. We also include a reference installation from source to help guide you through installing TensorFlow with GPU support.

If you are already comfortable with using Pip/Conda, virtual environments, or installing from sources, feel free to simply use the official guide here:

https://www.tensorflow.org/versions/master/get_started/os_setup.html

Selecting an installation environment

Many pieces of software use libraries and packages that are maintained separately. For developers, this is a good thing, as it enables code reuse, and they can focus on creating new functionality instead of recreating code that is already available. However, there is a cost associated with doing this. If a program depends on having another library available in order to work properly, the user or software must ensure that any machine running its code has that library installed. At first glance, this may seem like a trivial problem- simply install the required dependencies along with your software, right? Unfortunately, this approach could have some unintended consequences, and frequently does.

Imagine the following scenario: You find an awesome piece of software, Software A, so you download and install it. As part of its installation script, Software A looks for another piece of software that it depends on and installs it if your computer doesn't have it. We'll call that software Dependency, which is currently on version 1.0. Software A installs Dependency 1.0, finishes its own installation, and all is well. Some time in the future, you stumble upon another program you'd like to have, Software B. Software B uses Dependency 2.0, which is a complete overhaul from Dependency 1.0, and is not backwards compatible. Because of the way Dependency is distributed, there is no way to have both version 1.0 and 2.0 running side-by-side, as this would cause ambiguity when using it (Both are imported as *Dependency*: which version should be used?). Software B overwrites Dependency 1.0 with version 2.0 and completes its install. You find out (the hard way) that Software A is not compatible with Dependency 2.0, and is completely broken. All is not well. How can we run both Software A and Software B on the same machine? It's important for us to know, as TensorFlow depends on several open pieces of software. With Python (the language that packages TensorFlow), there are a couple of ways to get around this dependency clashing, as you'll see next.

- 1. Package dependencies inside of codebase:** Instead of relying on a system-level package or library, developers can choose to put the exact version of the library inside of their own code and reference it locally. In this way, all of the software's required code is available and won't be affected by external changes. This is not without its own downsides, however. First, it increases the disk space required to install the software, which means it takes longer to install and becomes more costly to use. Second, the user could have the dependency installed globally anyway, which means that the local version is redundant and eating up space. Finally, it's possible that the dependency puts out a critical, backwards compatible update, which fixes a serious security vulnerability. It now becomes the software developer's responsibility to update the dependency in their codebase, instead of having the user update it from a package manager. Unfortunately, the end-user doesn't have a lot of say with this method, as it's up to the developer to decide when to include dependencies directly. For several of its dependencies, TensorFlow does not include them, so they must be installed separately.
- 2. Use dependency environments:** Some package distribution managers have related

software that create environments, inside of which specific versions of software can be maintained independently of those contained in other environments. With Python, there are a couple of options. For the standard distributions of Python, Virtualenv is available. If you are using [Anaconda](#), it comes with a built-in environment system with its package manager, Conda. We'll cover how to install TensorFlow using both of these below.

3. **Use containers:** Containers, such as [Docker](#), are lightweight ways to package software with an entire file system, including its runtime and dependencies. Because of this, any machine (including virtual machines) that can run the container will be able to run the software identically to any other machine running that container. Starting up TensorFlow from a Docker container takes a few more steps than simply activating a Virtualenv or Conda environment, but its consistency across runtime environments can make it invaluable when deploying code across multiple instances (either on virtual machines or physical servers). We'll go over how to install Docker and create your own TensorFlow containers (as well as how to use the official TensorFlow image) below.

In general, we recommend using either Virtualenv or Conda's environments when installing TensorFlow for use on a single computer. They solve the conflicting dependency issue with relatively low overhead, are simple to setup, and require little thought once they are created. If you are preparing TensorFlow code to be deployed on one or more servers, it may be worth creating a Docker container image. While there are a few more steps involved, that cost pays itself back upon deployment across many servers. We do not recommend installing TensorFlow without using either an environment or container.

Jupyter Notebook and Matplotlib

Two excellent pieces of software that are frequently incorporated in data science workflows are the Jupyter Notebook and matplotlib. These have been used in conjunction with NumPy for years, and TensorFlow's tight integration with NumPy allows users to take advantage of their familiar work patterns. Both are open source and use permissive BSD licenses.

The [Jupyter Notebook](#) (formerly the iPython Notebook) allows you to interactively write documents that include code, text, outputs, LaTeX, and other visualizations. This makes it incredibly useful for creating reports out of exploratory analysis, since you can show the code used to create your visualizations right next to your charts. You can also include Markdown cells provide richly formatted text to share your insight on your particular approach. Additionally, the Jupyter Notebook is fantastic for prototyping ideas, as you can go back and edit portions of your code and run it directly from the notebook. Unlike many other interactive Python environments that require you to execute code line-by-line, the Jupyter Notebook has you write your code into logical chunks, which can make it easier to debug specific portions of your script. In TensorFlow, this is particularly useful, since a typical TensorFlow program is already split into “graph definition” and “graph running” portions.

[Matplotlib](#) is a charting library that allows you to create dynamic, custom visualizations in Python. It integrates seamlessly with NumPy, and its graphs can be displayed directly from the Jupyter Notebook. Matplotlib can also be used to display numeric data as images, which can be used for verifying outputs for image recognition tasks as well as visualizing internal components of neural networks. Additional layers on top of matplotlib, such as [Seaborn](#), can be used to augment its capabilities.

Creating a Virtualenv environment

To keep our dependencies nice and clean, we're going to be using [virtualenv](#) to create a virtual Python environment. First, we need to make sure that Virtualenv is installed along with pip, Python's package manager. Run the following commands (depending on which operating system you are running):

Linux 64-bit

```
# Python 2.7
$ sudo apt-get install python-pip python-dev python-virtualenv
# Python 3
$ sudo apt-get install python3-pip python3-dev python3-virtualenv
```

Mac OS X

```
$ sudo easy_install pip
$ sudo pip install --upgrade virtualenv
```

Now that we're ready to roll, let's create a directory to contain this environment, as well as any future environments you might create in the future:

```
$ sudo mkdir ~/env
```

Next, we'll create the environment using the `virtualenv` command. In this example, it will be located in `~/env/tensorflow`.

```
$ virtualenv --system-site-packages ~/env/tensorflow
```

Once it has been created, we can activate the environment using the `source` command.

```
$ source ~/env/tensorflow/bin/activate
# Notice that your prompt now has a '(tensorflow)' indicator
(tensorflow)$
```

We'll want to make sure that the environment is active when we install anything with `pip`, as that is how Virtualenv keeps track of various dependencies.

When you're done with the environment, you can shut it off just by using the `deactivate` command:

```
(tensorflow)$ deactivate
```

Since you'll be using the virtual environment frequently, it will be useful to create a shortcut for activating it instead of having to write out the entire `source...` command each time. This next command adds a bash alias to your `~/.bashrc` file, which will let you simply type `tensorflow` whenever you want to start up the environment:

```
$ sudo printf '\nalias tensorflow="source ~/env/tensorflow/bin/activate"' >> ~/.bashrc
```

To test it out, restart your bash terminal and type `tensorflow`:

```
$ tensorflow
# The prompt should change, as before
(tensorflow)$
```

Simple installation of TensorFlow

If you just want to get on to the tutorials as quickly as possible and don't care about GPU support, you can install one of TensorFlow's official pre-built binaries. Simply make sure that your Virtualenv environment from the previous section is active and run the following command corresponding to your operating system and version of Python:

Linux 64-bit installation

```
# Linux, Python 2.7
(tensorflow)$ pip install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.9.0-py2-none-any.whl

# Linux 64-bit, Python 3.4
(tensorflow)$ pip3 install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.9.0-py3-none-any.whl

# Linux 64-bit, Python 3.5
(tensorflow)$ pip3 install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.9.0-py3-none-any.whl
```

Mac OS X installation

```
# Mac OS X, Python 2.7:
(tensorflow)$ pip install --upgrade https://storage.googleapis.com/tensorflow/mac/tensorflow-0.9.0-py2-none-any.whl

# Mac OS X, Python 3.4+
(tensorflow)$ pip3 install --upgrade https://storage.googleapis.com/tensorflow/mac/tensorflow-0.9.0-py3-none-any.whl
```

Technically, there are pre-built binaries for TensorFlow with GPU support, but they require specific versions of NVIDIA software and are incompatible with future versions.

Example installation from source: 64-bit Ubuntu Linux with GPU support

If you want to use TensorFlow with GPU(s) support, you will most likely have to build from source. We've included a reference installation example that goes step-by-step through all of the things you'll need to do to get TensorFlow up and running. Note that this example is for an Ubuntu Linux 64-bit distribution, so you may have to change certain commands (such as `apt-get`). If you'd like to build from source on Mac OS X, we recommend the official guide on the TensorFlow website:

https://www.tensorflow.org/versions/master/get_started/os_setup.html#installation-for-mac-os-x

Installing dependencies

This assumes you've already installed `python-pip`, `python-dev`, and `python-virtualenv` from the previous section on installing `Virtualenv`.

Building TensorFlow requires a few more dependencies, though! Run the following commands, depending on your version of Python:

Python 2.7

```
$ sudo apt-get install python-numpy python-wheel python-imaging swig
```

Python 3

```
$ sudo apt-get install python3-numpy python3-wheel python3-imaging swig
```

Installing Bazel

[Bazel](#) is an open source build tool based on Google’s internal software, Blaze. As of writing, TensorFlow requires Bazel in order to build from source, so we must install it ourselves. The Bazel website has [complete installation instructions](#), but we include the basic steps here.

The first thing to do is ensure that Java Development Kit 8 is installed on your system. The following commands will add the Oracle JDK 8 repository as a download location for `apt` and then install it:

```
$ sudo apt-get install software-properties-common  
$ sudo add-apt-repository ppa:webupd8team/java  
$ sudo apt-get update  
$ sudo apt-get install oracle-java8-installer
```

Ubuntu versions 15.10 and later can install OpenJDK 8 instead of the Oracle JDK. This is easier and recommended- use the following commands instead of the above to install OpenJDK on your system:

```
# Ubuntu 15.10  
$ sudo apt-get install openjdk-8-jdk  
# Ubuntu 16.04  
$ sudo apt-get install default-jdk
```

Before moving on, verify that Java is installed correctly:

```
$ java -version  
# Should see similar output as below  
java version "1.8.0_91"  
Java(TM) SE Runtime Environment (build 1.8.0_91-b14)  
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b14, mixed mode)
```

Once Java is setup, there are a few more dependencies to install:

```
$ sudo apt-get install pkg-config zip g++ zlib1g-dev unzip
```

Next, you’ll need to download the Bazel installation script. To do so, you can either go to the [Bazel releases page](#) on GitHub, or you can use the following `wget` command. Note that for Ubuntu, you’ll want to download “`bazel — installer-linux-x86_64.sh`”:

```
# Downloads Bazel 0.3.0  
$ wget https://github.com/bazelbuild/bazel/releases/download/0.3.0/bazel-0.3.0-installer-linux-x86_6
```

Finally, we’ll make the script executable and run it:

```
$ chmod +x bazel-<version>-installer-linux-x86_64.sh  
$ ./bazel-<version>-installer-linux-x86_64.sh --user
```

By using the `--user` flag, Bazel is installed to the `/bin` directory for the user. To ensure that this is added to your `PATH`, run the following to update your `/.bashrc`:

```
$ sudo printf '\nexport PATH="$PATH:$HOME/bin"' >> ~/.bashrc
```

Restart your bash terminal and run `bazel` to make sure everything is working properly:

```
$ bazel version  
# You should see some output like the following  
Build label: 0.3.0  
Build target: ...  
...
```

Great! Next up, we need to get the proper dependencies for GPU support.

Installing CUDA Software (NVIDIA CUDA GPUs only)

If you have an NVIDIA GPU that supports [CUDA](#), you can install TensorFlow with GPU support. There is a list of CUDA-enabled video cards available here:

<https://developer.nvidia.com/cuda-gpus>

In addition to making sure that your GPU is on the list, make a note of the “Compute Capability” number associated with your card. For example, the GeForce GTX 1080 has a compute capability of 6.1, and the GeForce GTX TITAN X has a compute capability of 5.2. You’ll need this number for when you compile TensorFlow. Once you’ve determined that you’re able to take advantage of CUDA, the first thing you’ll want to do is sign up for NVIDIA’s “Accelerated Computer Developer Program”. This is required to download all of the files necessary to install CUDA and cuDNN. The link to do so is here:

<https://developer.nvidia.com/accelerated-computing-developer>

Once you’re signed up, you’ll want to download CUDA. Go to the following link and use the following instructions:

<https://developer.nvidia.com/cuda-downloads>

The screenshot shows the CUDA Download page. The top section, "Select Target Platform", is a grid of green buttons with some highlighted in red. The columns are labeled: Operating System (Windows, Linux, Mac OSX), Architecture (x86_64, ppc64le), Distribution (Fedora, OpenSUSE, RHEL, CentOS, SLES, SteamOS), Version (Ubuntu 15.04, 14.04), and Installer Type (runfile (local), deb (local), deb (network)). The "Linux" button is highlighted. The "14.04" button is highlighted. The "deb (local)" button is highlighted. The right side of the page has a "Documentation" sidebar with links to CUDA Quick Start Guide, Release Notes, EULA, Online Documentation, CUDA Toolkit Overview, Download Checksums, and Legacy Toolkits.

Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown.

Operating System	Windows	Linux	Mac OSX			
Architecture	x86_64	ppc64le				
Distribution	Fedora	OpenSUSE	RHEL	CentOS	SLES	SteamOS
Version	Ubuntu 15.04	Ubuntu 14.04				
Installer Type	runfile (local)	deb (local)	deb (network)			

Documentation

- CUDA Quick Start Guide
- Release Notes
- EULA
- Online Documentation
- CUDA Toolkit Overview
- Download Checksums
- Legacy Toolkits

Download Target Installer for Linux Ubuntu 14.04 x86_64

cuda-repo-ubuntu1404-7-5-local_7.5-18_amd64.deb (md5sum: 5cf65b8139d70270d9234d5ff4d697c7)

Download (1.9 GB)

Installation Instructions:

1. `sudo dpkg -i cuda-repo-ubuntu1404-7-5-local_7.5-18_amd64.deb`
2. `sudo apt-get update`
3. `sudo apt-get install cuda`

For further information, see the [Installation Guide for Linux](#) and the [CUDA Quick Start Guide](#).

1. Under “Select Target Platform”, choose the following options:

- Linux
- x86_64
- Ubuntu

- 14.04/15.04 (whichever version you are using)
- deb (local)

2. Click the “Download” button and save it somewhere on your computer. This file is large, so it will take a while.
3. Navigate to the directory containing the downloaded file and run the following commands:

```
$ sudo dpkg -i cuda-repo-ubuntu1404-7-5-local_7.5-18_amd64.deb
$ sudo apt-get update
$ sudo apt-get install cuda
```

This will install CUDA into the `/usr/local/cuda` directory.

Next, we need to download cuDNN, which is a separate add-on to CUDA designed for deep neural networks. Click the “Download” button on the following page:

<https://developer.nvidia.com/cudnn>

After signing in with the account you created above, you’ll be taken to a brief survey. Fill that out and you’ll be taken to the download page. Click “I Agree to the Terms...” to be presented with the different download options. Because we installed CUDA 7.5 above, we’ll want to download cuDNN for CUDA 7.5 (as of writing, we are using cuDNN version 5.0).

Click “Download cuDNN v5 for CUDA 7.5” to expand a bunch of download options:

Home > ComputeWorks > Deep Learning > Software > cuDNN Download

cuDNN Download

NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks.

I Agree To the Terms of the [cuDNN Software License Agreement](#)

Please check your framework documentation to determine the recommended version of cuDNN. If you are using cuDNN with a Pascal (GTX 1080, GTX 1070), version 5 or later is required.

[Download cuDNN v5.1 RC \[June 19, 2016\], for CUDA 8.0 RC](#)

[Download cuDNN v5.1 RC \[June 16, 2016\], for CUDA 7.5](#)

[Download cuDNN v5 \[May 27, 2016\], for CUDA 8.0 RC](#)

[Download cuDNN v5 \[May 12, 2016\], for CUDA 7.5](#)

[Download cuDNN v4 \[Feb 10, 2016\], for CUDA 7.0 and later.](#)

[Archived cuDNN Releases](#)

QUICKLINKS
Accelerated Computing - Training
CUDA GPUs
Tools & Ecosystem
OpenACC: More Science Less Programming
CUDA FAQ

GPU Computing

[Follow](#)

 CUDA, GPU Computing Retweeted

 MapD
@datarefined

Not just `#machinelearning`, `#GPUs` are now crushing `#CPU` constrained solutions in traditional enterprise compute tasks
ow.ly/Zh6N301LeID



Click “cuDNN v5 Library for Linux” to download the zipped cuDNN files:

cuDNN Download

NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks.

I Agree To the Terms of the cuDNN Software License Agreement

Please check your framework documentation to determine the recommended version of cuDNN.
If you are using cuDNN with a Pascal (GTX 1080, GTX 1070), version 5 or later is required.

[Download cuDNN v5.1 RC \(June 19, 2016\), for CUDA 8.0 RC](#)

[Download cuDNN v5.1 RC \(June 16, 2016\), for CUDA 7.5](#)

[Download cuDNN v5 \(May 27, 2016\), for CUDA 8.0 RC](#)

[Download cuDNN v5 \(May 12, 2016\), for CUDA 7.5](#)

[cuDNN User Guide](#)

[cuDNN Install Guide](#)

[cuDNN v5 Library for Linux](#)

[cuDNN v5 Library for Linux \(IBM Power8\)](#)

[cuDNN v5 Library for Windows 7](#)

[cuDNN v5 Library for Windows 10](#)

[cuDNN v5 Library for OSX](#)

[cuDNN v5 Code Samples](#)

[cuDNN v5 Release Notes](#)

[cuDNN v5 Runtime Library for Linux \(Deb\)](#)

[cuDNN v5 Developer Library for Linux \(Deb\)](#)

[cuDNN v5 Code Samples and User Guide \(Deb\)](#)

QUICKLINKS

[Accelerated Computing - Training](#)

[CUDA GPUs](#)

[Tools & Ecosystem](#)

[OpenACC: More Science Less Programming](#)

[CUDA FAQ](#)

GPU Computing

[Follow](#)

 CUDA, GPU Computing Retweeted

 **MapD**
@datarefined

Not just #machinelearning, #GPUs are now crushing #CPU constrained solutions in traditional enterprise compute tasks
ow.ly/Zh6N301LErD



Navigate to where the .tgz file was downloaded and run the following commands to place the correct files inside the /usr/local/cuda directory:

```
$ tar xvzf cudnn-7.5-linux-x64-v5.0-ga.tgz
$ sudo cp cuda/include/cudnn.h /usr/local/cuda/include
$ sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
$ sudo chmod a+r /usr/local/cuda/include/cudnn.h /usr/local/cuda/lib64/libcudnn*
```

And that's it for installing CUDA! With all of the dependencies taken care of, we can now move on to the installation of TensorFlow itself.

Building and Installing TensorFlow from Source

First things first, clone the Tensorflow repository from GitHub and enter the directory:

```
$ git clone --recurse-submodules https://github.com/tensorflow/tensorflow
$ cd tensorflow
```

Once inside, we need to run the `./configure` script, which will tell Bazel which compiler to use, which version of CUDA to use, etc. Make sure that you have the “compute capability” number (as mentioned previously) for your GPU card available:

```
$ ./configure
Please specify the location of python. [Default is /usr/bin/python]: /usr/bin/python
# NOTE: For Python 3, specify /usr/bin/python3 instead
Do you wish to build TensorFlow with Google Cloud Platform support? [y/N] N
Do you wish to build TensorFlow with GPU support? [y/N] y

Please specify which gcc nvcc should use as the host compiler. [Default is /usr/bin/gcc]: /usr/bin/g
Please specify the Cuda SDK version you want to use, e.g. 7.0. [Leave empty to use system default]:
Please specify the Cudnn version you want to use. [Leave empty to use system default]: 5.0.5
Please specify the location where cuDNN 5.0.5 library is installed. Refer to README.md for more data
Please specify a list of comma-separated Cuda compute capabilities you want to build with.
You can find the compute capability of your device at: https://developer.nvidia.com/cuda-gpus.
Please note that each additional compute capability significantly increases your build time and binary size.
[Default is: "3.5,5.2"]:<YOUR-COMPUTE-CAPABILITY-NUMBER-HERE>

Setting up Cuda include
Setting up Cuda lib64
Setting up Cuda bin
Setting up Cuda nvvm
Setting up CUPTI include
Setting up CUPTI lib64
Configuration finished
```

Google Cloud Platform support is currently in a closed alpha. If you have access to the program, feel free to answer yes to the Google Cloud Platform support question.

With the configuration finished, we can use Bazel to create an executable that will create our Python binaries:

```
$ bazel build -c opt --config=cuda //tensorflow/tools/pip_package:build_pip_package
```

This will take a fair amount of time, depending on how powerful your computer is. Once Bazel is done, run the output executable and pass in a location to save the Python wheel:

```
$ bazel-bin/tensorflow/tools/pip_package/build_pip_package ~/tensorflow/bin
```

This creates a Python .whl file inside of `~/tensorflow/bin/`. Make sure that your “tensorflow” Virtualenv is active, and install the wheel with pip! (Note that the exact name of the binary will differ depending on which version of TensorFlow is installed, which operating system you’re using, and which version of Python you installed with):

```
$ tensorflow
(tensorflow)$ sudo pip install ~/tensorflow/bin/tensorflow-0.9.0-py2-none-any.whl
```

If you have multiple machines with similar hardware, you can use this wheel to quickly install TensorFlow on all of them.

You should be good to go! We'll finish up by installing the Jupyter Notebook and matplotlib.

Installing Jupyter Notebook:

First, run the following commands to install [iPython](#), an incredibly useful interactive Python kernel, and the backbone of the Jupyter Notebook. We highly recommend installing both the Python 2 and Python 3 kernels below, as it will give you more options moving forward (i.e., run all of the following!):

```
# Python 2.7
$ sudo python2 -m pip install ipykernel
$ sudo python2 -m ipykernel install
# Python 3
$ sudo python3 -m pip install jupyterhub notebook ipykernel
$ sudo python3 -m ipykernel install
```

After that, two simple commands should get you going. First install the `build-essential` dependency:

```
$ sudo apt-get install build-essential
```

Then use `pip` to install the Jupyter Notebook (`pip3` if you are using Python 3):

```
# For Python 2.7
$ sudo pip install jupyter
# For Python 3
$ sudo pip3 install jupyter
```

Official installation instructions are available at the Jupyter website:

<http://jupyter.readthedocs.io/en/latest/install.html>

Installing matplotlib

Installing matplotlib on Linux/Ubuntu is easy. Just run the following:

```
# Python 2.7
$ sudo apt-get build-dep python-matplotlib python-tk
# Python 3
$ sudo apt-get build-dep python3-matplotlib python3-tk
```

And that's it!

Testing Out TensorFlow, Jupyter Notebook, and matplotlib

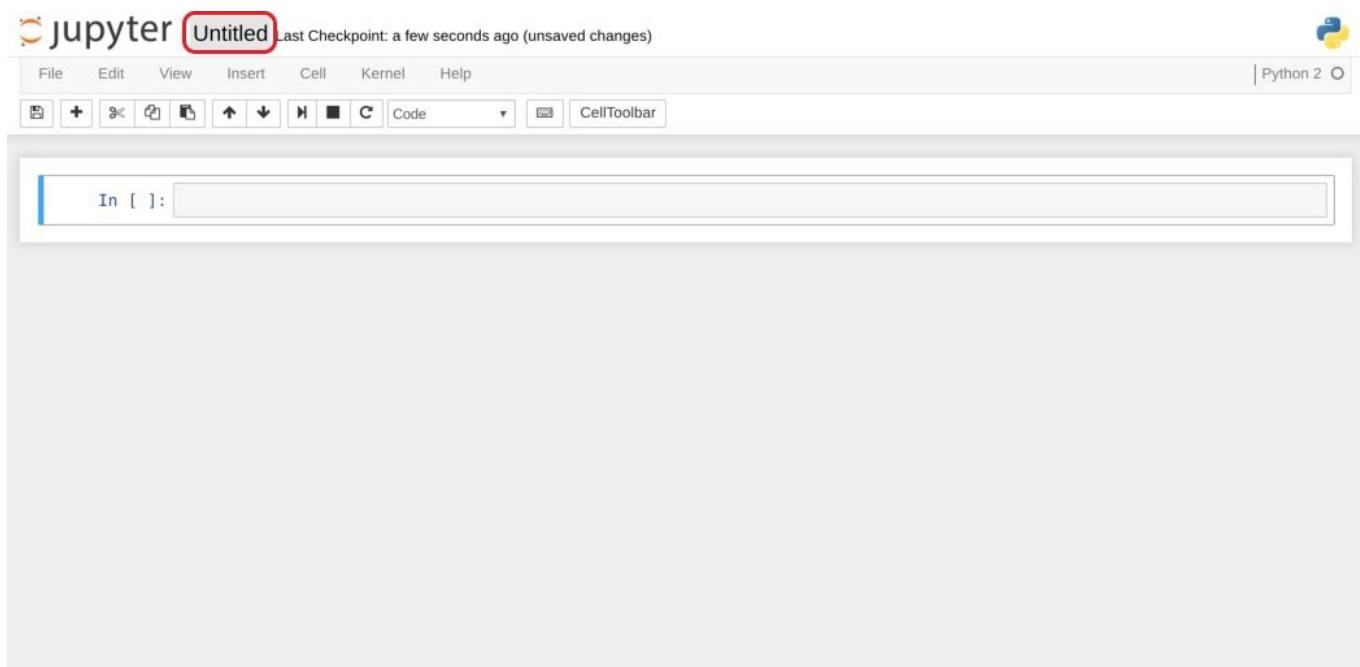
Let's run some dummy code to double check that things are working properly. Create a new directory called "tf-notebooks" to play around in. Enter that directory and run `jupyter notebook`. Again, make sure that the "tensorflow" environment is active:

```
(tensorflow)$ mkdir tf-notebooks  
(tensorflow)$ cd tf-notebooks  
(tensorflow)$ jupyter notebook
```

This will start up a Jupyter Notebook server and open the software up in your default web browser. Assuming you don't have any files in your `tf-notebooks` directory, you'll see an empty workspace with the message "Notebook list is empty". To create a new notebook, click the "New" button in the upper right corner of the page, and then select either "Python 2" or "Python 3", depending on which version of Python you've used to install TensorFlow.

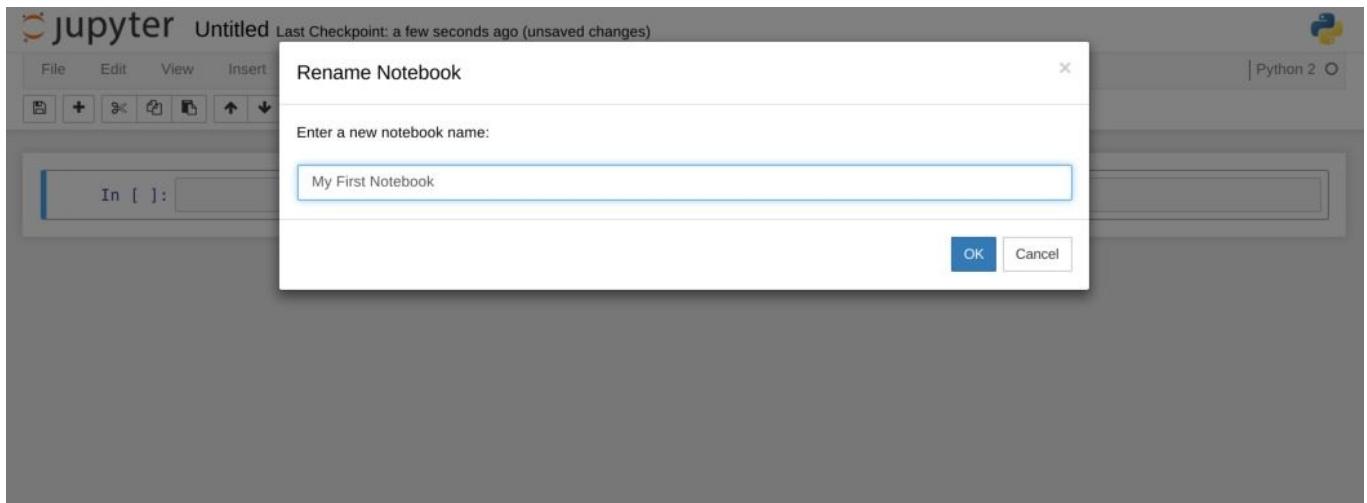


Your new notebook will open up automatically, and you'll be presented with a blank slate to work with. Let's quickly give the notebook a new name. At the top of the screen, click the word "Untitled":



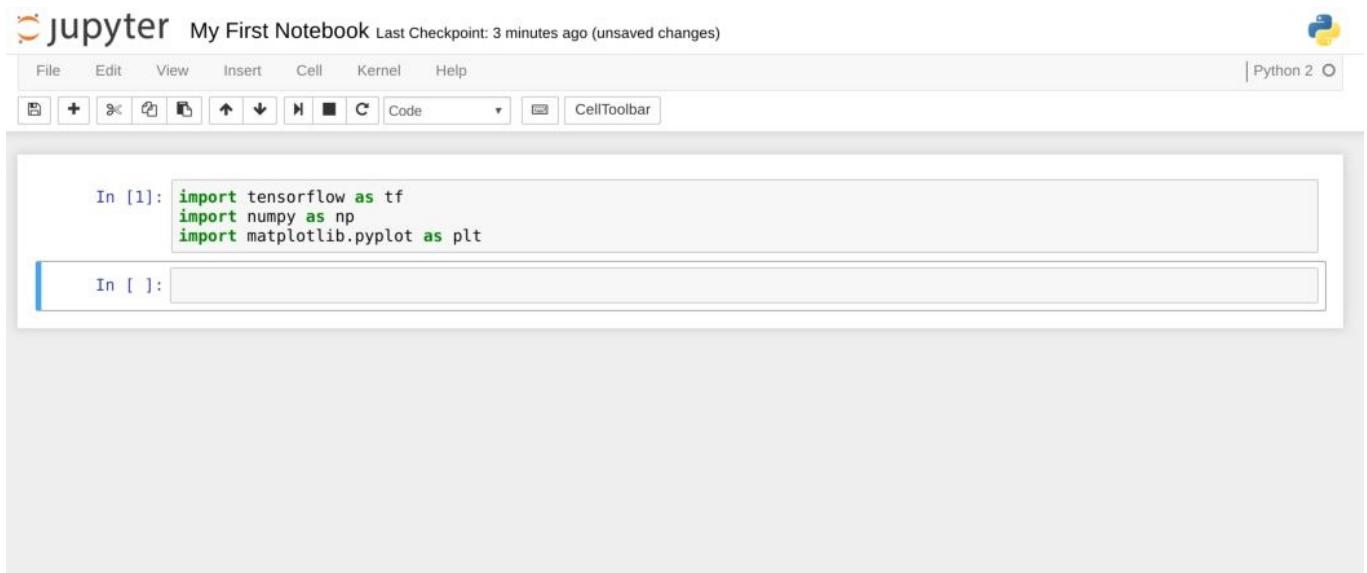
This will pop up a window that allows you to rename the notebook. This also changes

the name of the notebook file (with the extension .ipynb). You can call this whatever you'd like- in this example we're calling it "My First Notebook"



Now, let's look at the actual interface. You'll notice an empty cell with the block `In []:` next to it. You can type Python code directly into this cell, and it can include multiple lines. Let's import TensorFlow, NumPy, and the pyplot module of matplotlib into notebook:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```



In order to run the cell, simply type *shift-enter*, which will run your code and create a new cell below. You'll notice the indicator to the left now reads `In [1]:`, which means that this cell was the first block of code to run in the kernel. Fill in the notebook with the following code, using as many or as few cells as you find appropriate. You can use the breaks in the cells to naturally group related code together.

```
%matplotlib inline
a = tf.random_normal([2, 20])
sess = tf.Session()
out = sess.run(a)
x, y = out

plt.scatter(x, y)
plt.show()
```

Jupyter My First Notebook Last Checkpoint: 9 minutes ago (unsaved changes) Python 2

File Edit View Insert Cell Kernel Help

Code CellToolbar

```
In [1]: import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

In [2]: %matplotlib inline

In [3]: a = tf.random_normal([2,20])

In [4]: sess = tf.Session()

In [5]: out = sess.run(a)

In [6]: x, y = out

In [7]: plt.scatter(x, y)
plt.show()
```

This line is special, and worth mentioning in particular:

```
%matplotlib inline
```

It is a special command that tells the notebook to display matplotlib charts directly inside the browser.

Let's go over what the rest of the code does, line-by-line. Don't worry if you don't understand some of the terminology, as we'll be covering it in the book:

1. Use TensorFlow to define a 2×20 matrix of random numbers and assign it to the variable `a`
2. Start a TensorFlow Session and assign it to `sess`
3. Execute `a` with the `sess.run()` method, and assign the output (which is a NumPy array) to `out`
4. Split up the 2×20 matrix into two 1×10 vectors, `x` and `y`
5. Use `pyplot` to create a scatter plot with `x` and `y`

Assuming everything is installed correctly, you should get an output similar to the above! It's a small first step, but hopefully it feels good to get the ball rolling.

For a more thorough tutorial on the ins-and-outs of the Jupyter Notebook, check out the examples page here:

http://jupyter-notebook.readthedocs.io/en/latest/examples/Notebook/examples_index.html

Conclusion

Voila! You should have a working version of TensorFlow ready to go. In the next chapter, you'll learn fundamental TensorFlow concepts and build your first models in the library. If you had any issues installing TensorFlow on your system, the official installation guide should be the first place to look:

https://www.tensorflow.org/versions/master/get_started/os_setup.html

Part II. TensorFlow and Machine Learning fundamentals

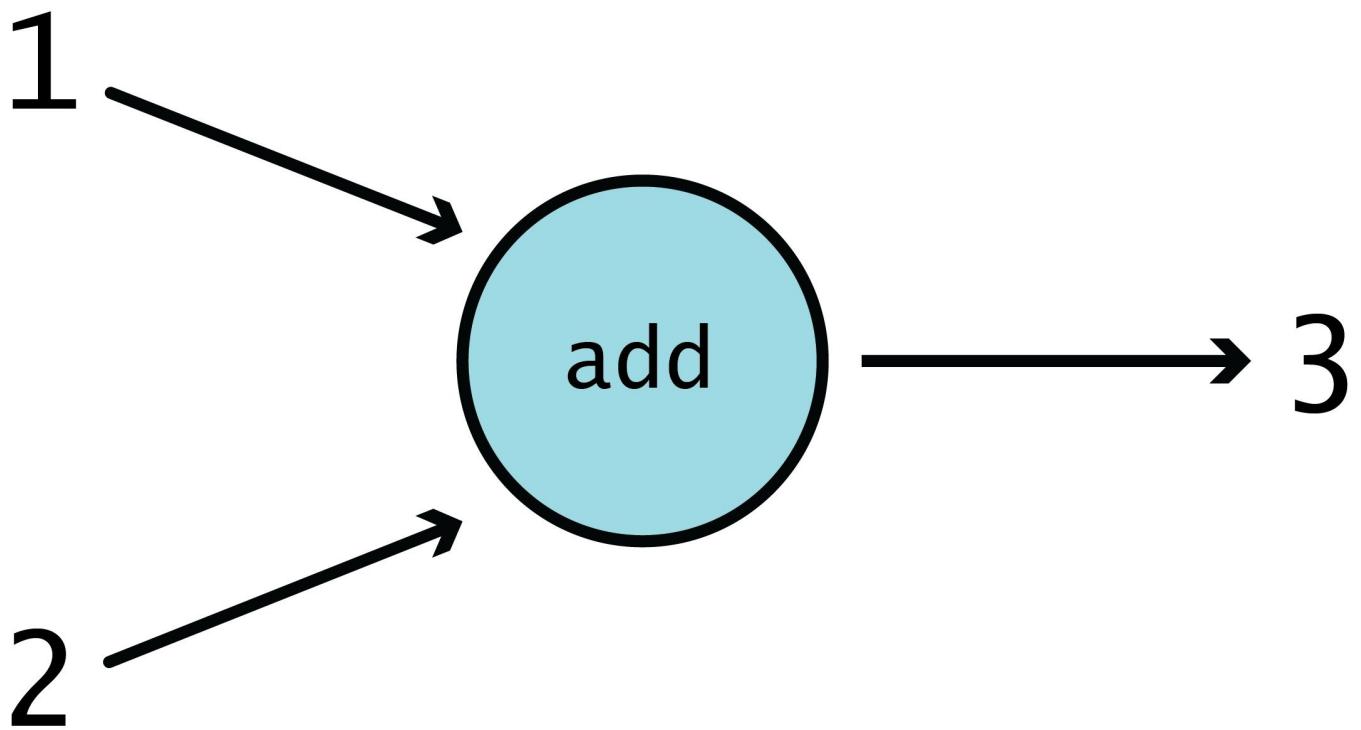
Chapter 3. TensorFlow Fundamentals

Introduction to Computation Graphs

This section covers the basics of computation graphs without the context of TensorFlow. This includes defining nodes, edges, and dependencies, and we also provide several examples to illustrate key principles. If you are experienced and/or comfortable with computation graphs, you may skip to the next section.

Graph basics

At the core of every TensorFlow program is the *computation graph* described in code with the TensorFlow API. A computation graph, is a specific type of directed graph that is used for defining, unsurprisingly, computational structure. In TensorFlow it is, in essence, a series of functions chained together, each passing its output to zero, one, or more functions further along in the chain. In this way, a user can construct a complex transformation on data by using blocks of smaller, well-understood mathematical functions. Let's take a look at a bare-bones example.



In the above example, we see the graph for basic addition. The function, represented by a circle, takes in two inputs, represented as arrows pointing into the function. It outputs the result of adding 1 and 2 together: 3, which is shown as an arrow pointing out of the function. The result could then be passed along to another function, or it might simply be returned to the client.

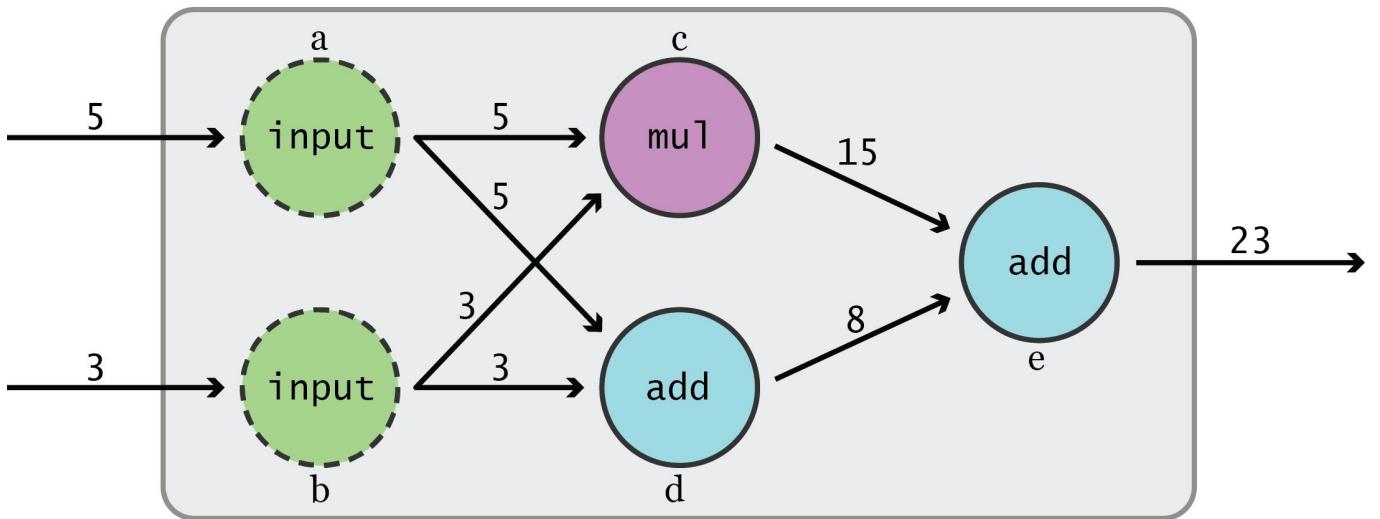
We can also look at this graph as a simple equation:

$$f(1, 2) = 1 + 2 = 3$$

The above illustrates how the two fundamental building blocks of graphs, nodes and edges, are used when constructing a computation graph. Let's go over their properties:

- **Nodes**, typically drawn as circles, ovals, or boxes, represent some sort of *computation* or *action* being done on or with data in the graph's context. In the above example, the operation "add" is the sole node.
- **Edges** are the actual values that get passed to and from Operations, and are typically drawn as arrows. In the "add" example, the inputs 1 and 2 are both edges leading into the node, while the output 3 is an edge leading out of the node. Conceptually, we can think of edges as the link between different Operations as they carry information from one node to the next.

Now, here's a slightly more interesting example:



There's a bit more going on in this graph! The data is traveling from left to right (as indicated by the direction of the arrows), so let's break down the graph, starting from the left.

1. At the very beginning, we can see two values flowing into the graph, 5 and 3. They may be coming from a different graph, being read in from a file, or entered directly by the client.
2. Each of these initial values is passed to one of two explicit “input” nodes, labeled a and b in the graphic. The “input” nodes simply pass on values given to them- node a receives the value 5 and outputs that same number to nodes c and d , while node b performs the same action with the value 3.
3. Node c is a multiplication operation. It takes in the values 5 and 3 from nodes a and b , respectively, and outputs its result of 15 to node e . Meanwhile, node d performs addition with the same input values and passes the computed value of 8 along to node e .
4. Finally, node e , the final node in our graph, is another “add” node. It receives the values of 15 and 8, adds them together, and spits out 23 as the final result of our graph.

Here's how the above graphical representation might look as a series of equations:

$$\begin{aligned}
 a &= \text{input}_1; & b &= \text{input}_2 \\
 c &= a \cdot b; & d &= a + b \\
 e &= c + d
 \end{aligned}$$

If we wanted to solve e for $a = 5$ and $b = 3$, we can just work backwards from e and plug in!

$$a = 5; b = 3$$

$$e = (a \cdot b) + (a + b)$$

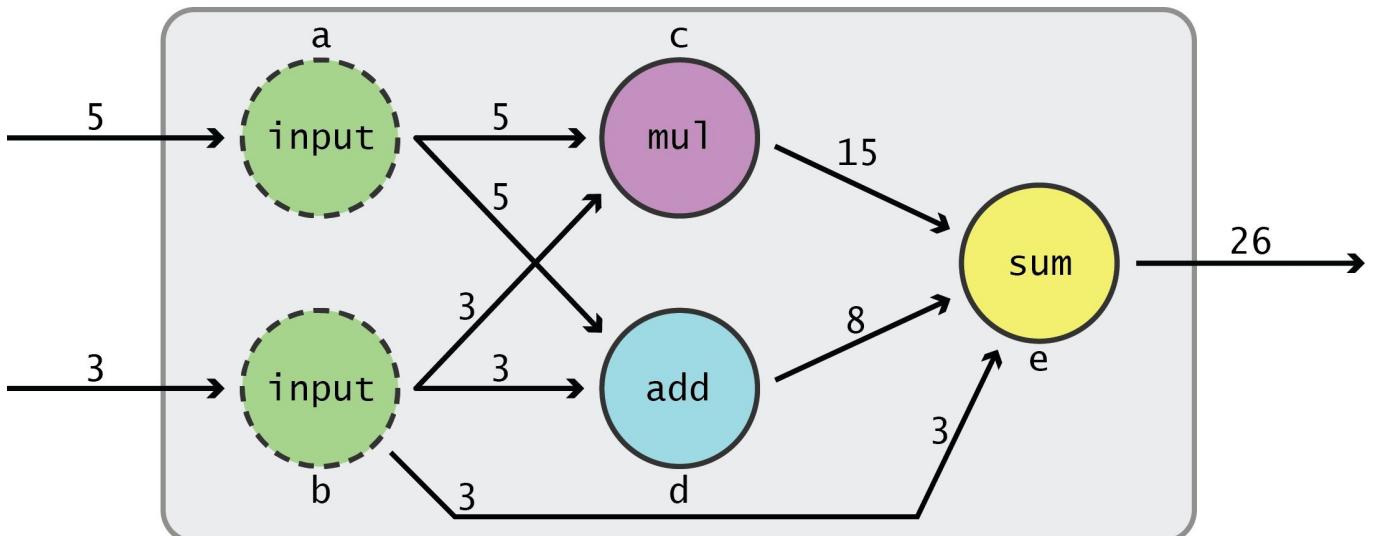
$$e = (5 \cdot 3) + (5 + 3)$$

$$e = 15 + 8 = 23$$

With that, the computation is complete! There are concepts worth pointing out here:

- The pattern of using “input” nodes is useful, as it allows us to relay a single input value to a huge amount of future nodes. If we didn’t do this, the client (or whoever passed in the initial values) would have to explicitly pass each input value to multiple nodes in our graph. This way, the client only has to worry about passing in the appropriate values once and any repeated use of those inputs is abstracted away. We’ll touch a little more on abstracting graphs shortly.
- Pop quiz: which node will run first- the multiplication node c , or the addition node d ? The answer: you can’t tell. From just this graph, it’s impossible to know which of c and d will execute first. Some might read the graph from left-to-right *and* top-to-bottom and simply assume that node c would run first, but it’s important to note that the graph could have easily been drawn with d on top of c . Others may think of these nodes as running concurrently, but that may not always be the case, due to various implementation details or hardware limitations. In reality, it’s best to think of them as running *independently* of one another. Because node c doesn’t rely on any information from node d , it doesn’t have to wait for node d to do anything in order to complete its operation. The converse is also true: node d doesn’t need any information from node c . We’ll talk more about dependency later in this chapter.

Next, here’s a slightly modified version of the graph:

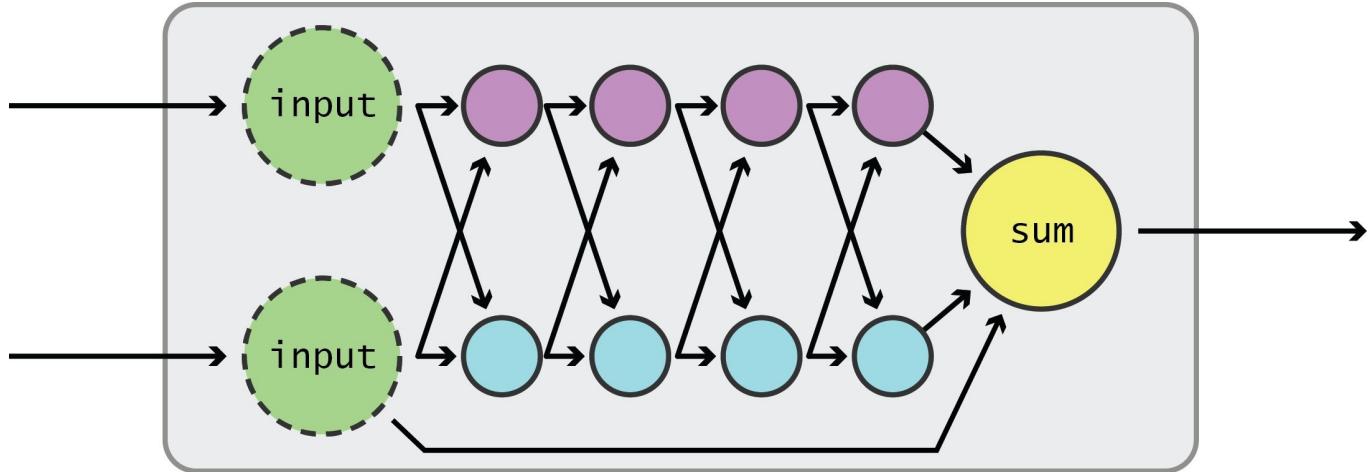


There are two main changes here:

1. The “input” value 3 from node b is now being passed on to node e .
2. The function “add” in node e has been replaced with “sum”, to indicate that it adds

more than two numbers.

Notice how we are able to add an edge between nodes that appear to have other nodes “in the way.” In general, any node can pass its output to any future node in the graph, no matter how many computations take place in between. The graph could have looked like the following, and still be perfectly valid:



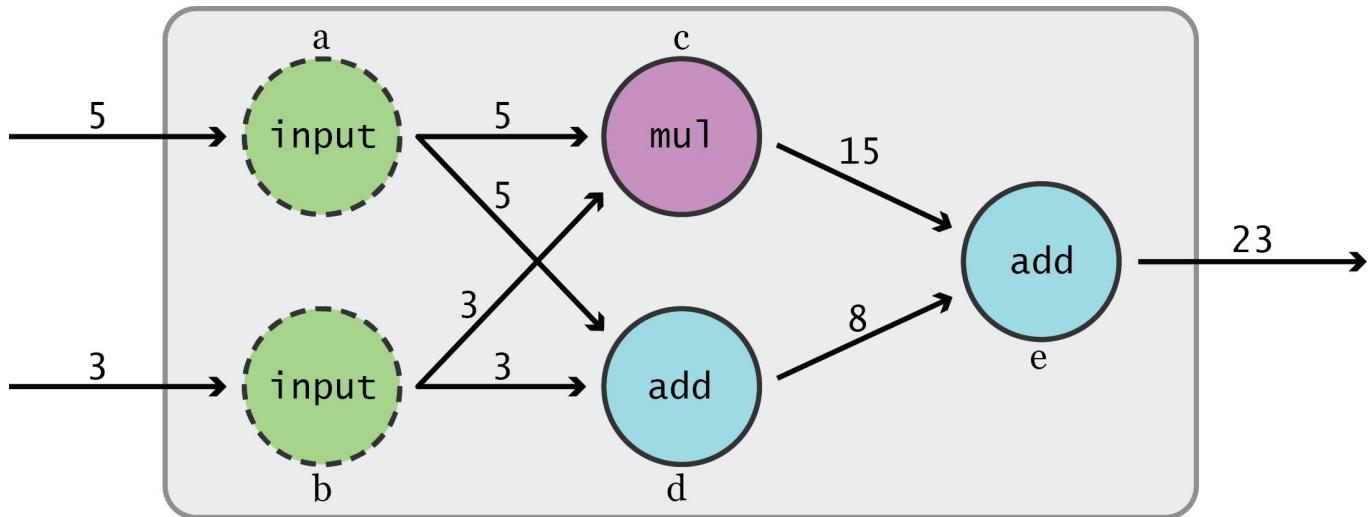
With both of these graphs, we can begin to see the benefit of abstracting the graph’s input. We were able to manipulate the precise details of what’s going on inside of our graph, but the client only has to know to send information to the same two input nodes. We can extend this abstraction even further, and can draw our graph like this:



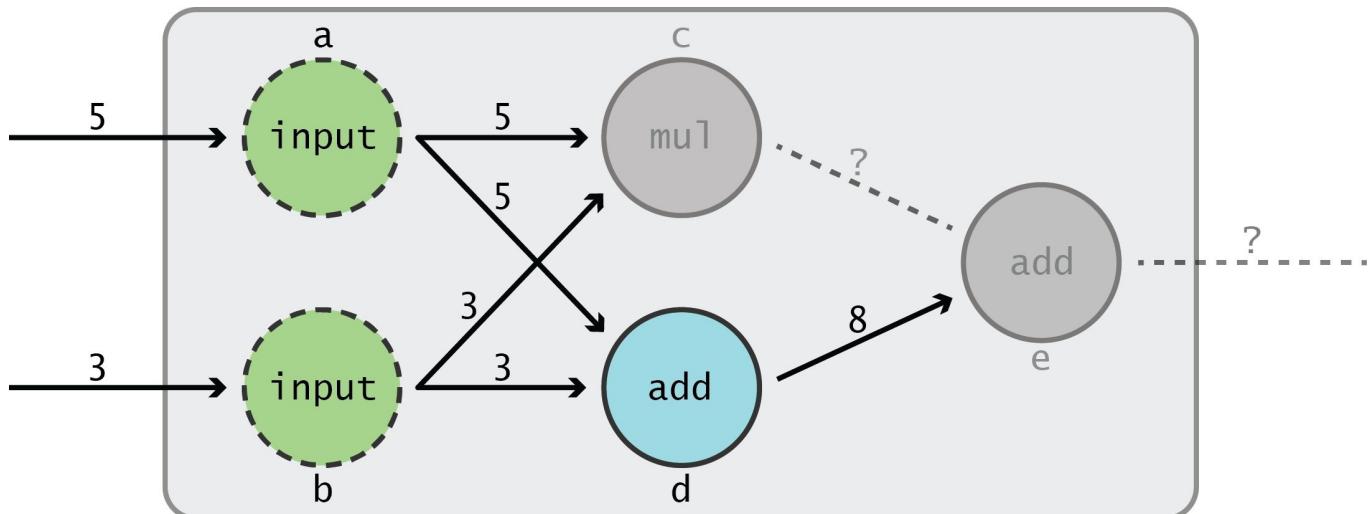
By doing this we can think of entire sequences of nodes as discrete building blocks with a set input and output. It can be easier to visualize chaining together groups of computations instead of having to worry about the specific details of each piece.

Dependencies

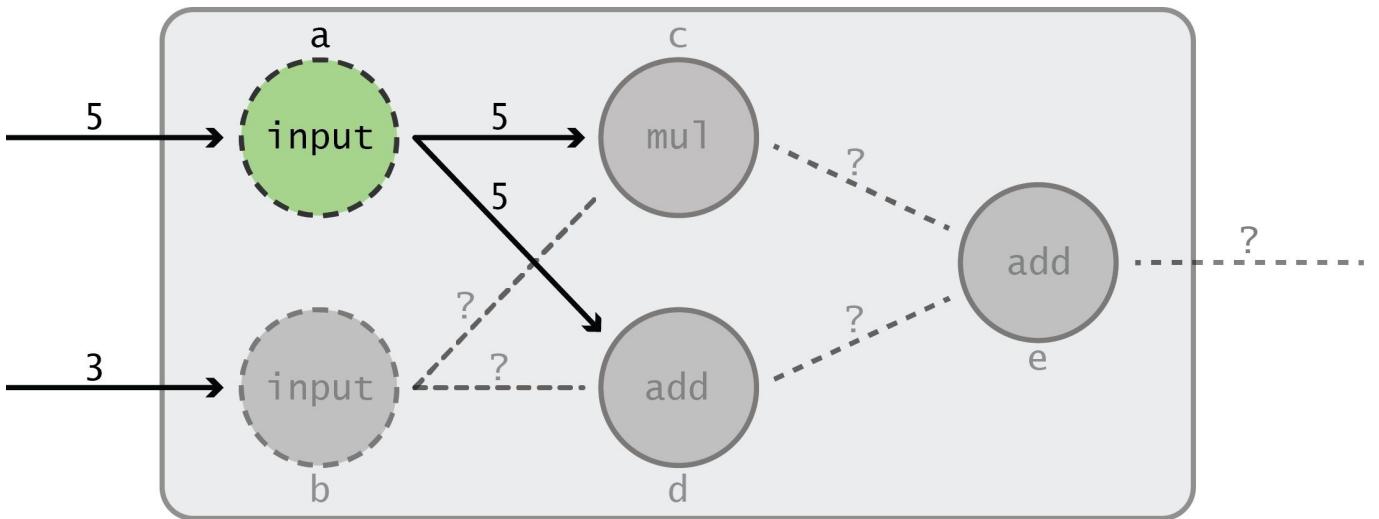
There are certain types of connections between nodes that aren't allowed, the most common of which is one that creates an unresolved *circular dependency*. In order to explain a circular dependency, we're going to illustrate what a dependency is. Let's take a look at this graph again:



The concept of a dependency is straight-forward: any node, A, that is required for the computation of a later node, B, is said to be a *dependency* of B. If a node A and node B do not need any information from one another, they are said to be *independent*. To visually represent this, let's take a look at what happens if the multiplication node c is unable to finish its computation (for whatever reason):



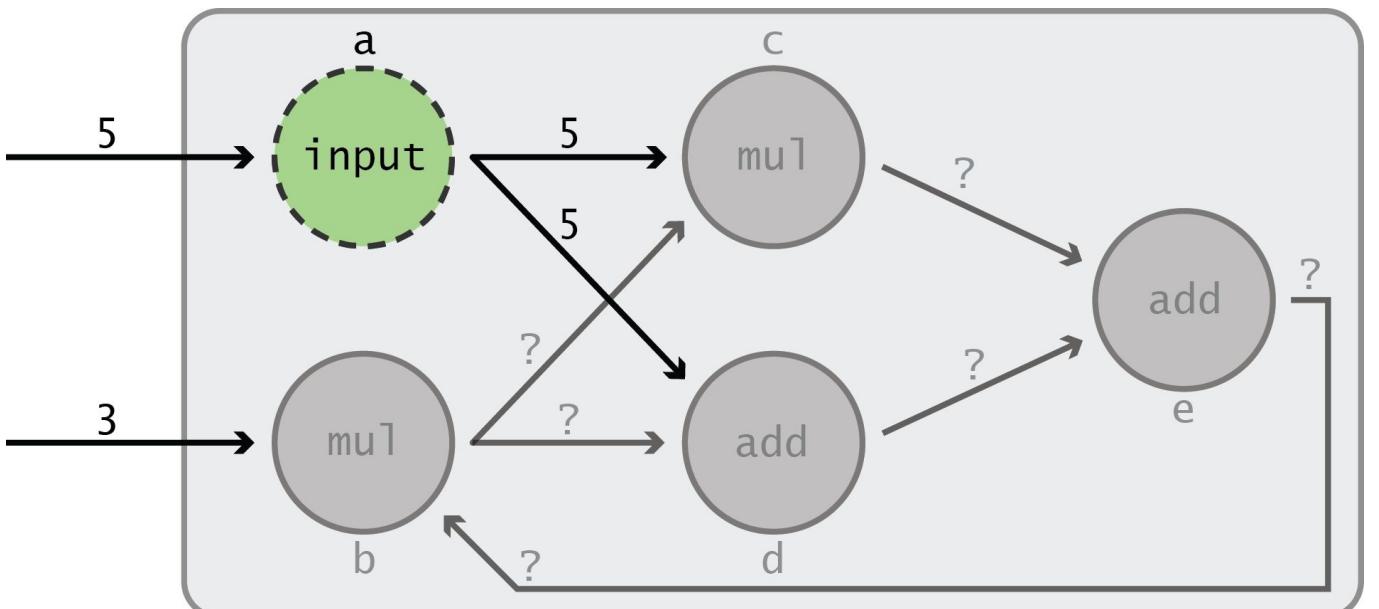
Predictably, since node e requires the output from node c, it is unable to perform its calculation and waits indefinitely for node c's data to arrive. It's pretty easy to see that nodes c and d are dependencies of node e, as they feed information directly into the final addition function. However, it may be slightly less obvious to see that the inputs a and b are also dependencies of e. What happens if one of the inputs fails to pass its data on to the next functions in the graph?



As you can see, removing one of the inputs halts most of the computation from actually occurring, and this demonstrates the *transitivity* of dependencies. That is to say, if A is dependent on B, and B is dependent on C, then A is dependent on C. In this case, the final node e is dependent on nodes c and d , and the nodes c and d are both dependent on input node b . Therefore, the final node e is dependent on the input node b . We can make the same reasoning for node e being dependent on node a , as well. Additionally, we can make a distinction between the different dependencies e has:

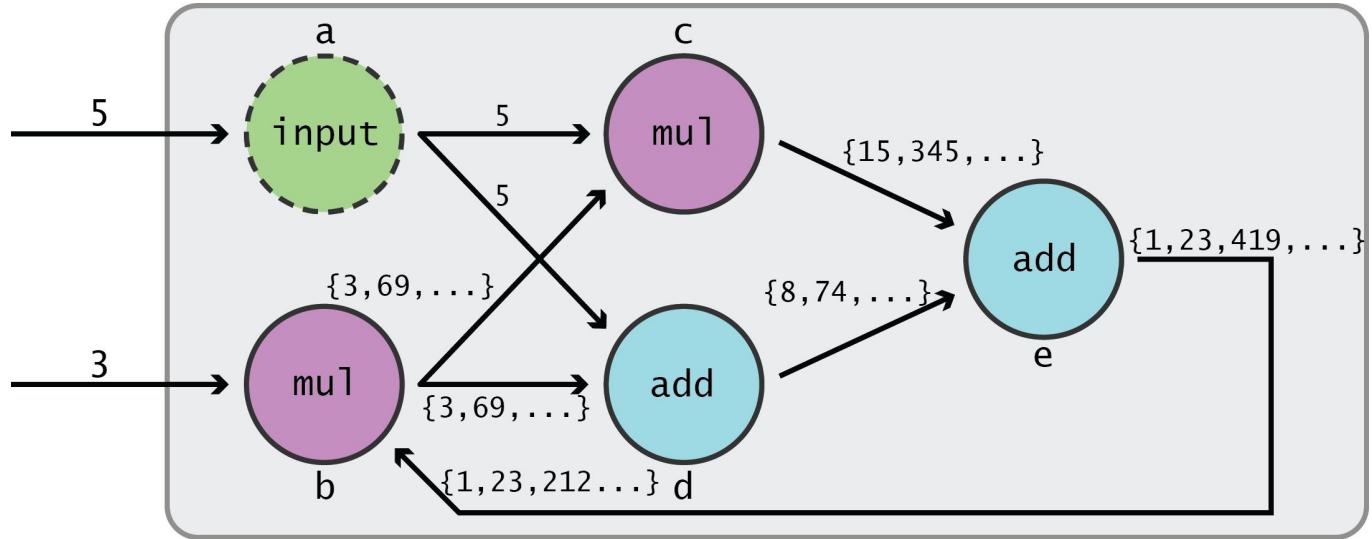
1. We can say that e is *directly dependent* on nodes c and d . By this, we mean that data must come *directly* from both node c and d in order for node e to execute.
2. We can say that e is *indirectly dependent* on nodes a and b . This means that the outputs of a and b do *not* feed directly into node e . Instead, their values are fed into an intermediary node(s) which is also a dependency of e , which can either be a direct dependency or indirect dependency. This means that a node can be indirectly dependent on a node with many layers of intermediaries in-between (and each of those intermediaries is also a dependency).

Finally, let's see what happens if we redirect the output of a graph back into an earlier portion of it:



Well, unfortunately it looks like that isn't going to fly. We are now attempting to pass the output of node `e` back into node `b` and, hopefully, have the graph cycle through its computations. The problem here is that node `b` now has node `e` as a direct dependency, while at the same time, node `e` is dependent on node `b` (as we showed previously). The result of this is that neither `b` nor `e` can execute, as they are both waiting for the other node to complete its computation.

Perhaps you are clever and decide that we could provide some initial state to the value feeding into either `b` or `e`. It is our graph, after all. Let's give the graph a kick-start by giving the output of `e` an initial value of 1:

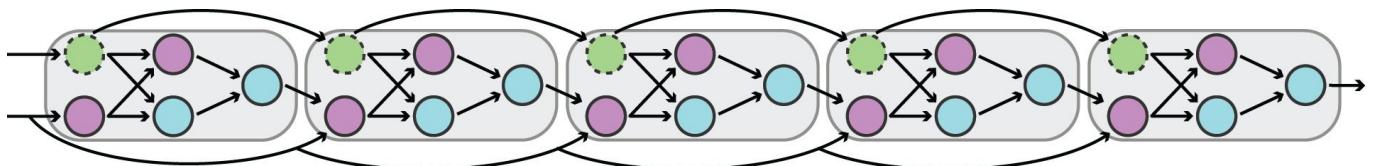


Here's what the first few loops through the graph look like. It creates an endless feedback loop, and most of the edges in the graph tend towards infinity. Neat! However, for software like TensorFlow, these sorts of infinite loops are bad for a number of reasons:

1. Because it's an infinite loop, the termination of the program isn't going to be graceful.
2. The number of dependencies becomes infinite, as each subsequent iteration is dependent on all previous iterations. Unfortunately, each node does not count as a single dependency- each time its output changes values it is counted again. This makes it impossible to keep track of dependency information, which is critical for a number of reasons (see the end of this section).
3. Frequently you end up in situations like this scenario, where the values being passed on either explode into huge positive numbers (where they will eventually overflow), huge negative numbers (where you will eventually underflow), or become close to zero (at which point each iteration has little additional meaning).

Because of this, truly circular dependencies can't be expressed in TensorFlow, which is not a bad thing. In practical use, we simulate these sorts of dependencies by copying a finite number of versions of the graph, placing them side-by-side, and feeding them into one another in sequence. This process is commonly referred to as "unrolling" the graph, and will be touched on more in the chapter on recurrent neural networks. To visualize what this unrolling looks like graphically, here's what the graph would look like after

we've unrolled this circular dependency 5 times:



If you analyze this graph, you'll discover that this sequence of nodes and edges is identical to looping through the previous graph 5 times. Note how the original input values (represented by the arrows skipping along the top and bottom of the graph) get passed onto each copy as they are needed for each copied "iteration" through the graph. By unrolling our graph like this, we can simulate useful cyclical dependencies while maintaining a deterministic computation.

Now that we understand dependencies, we can talk about why it's useful to keep track of them. Imagine for a moment, that we only wanted to get the output of node c from the previous example (the multiplication node). We've already defined the entire graph, including node d , which is independent of c , and node e , which occurs after c in the graph. Would we have to calculate the entire graph, even though we don't need the values of d and e ? No! Just by looking at the graph, you can see that it would be a waste of time to calculate all of the nodes if we only want the output from c . The question is: how do we make sure our computer only computes the necessary nodes without having to tell it by hand? The answer: use our dependencies!

The concept behind this is fairly simple, and the only thing we have to ensure is that each node has a list of the nodes it directly (*not* indirectly) depends on. We start with an empty stack, which will eventually hold all of the nodes we want to run. Start with the node(s) that you want to get the output from. Obviously it must execute, so we add it to our stack. We look at our output node's list of dependencies- which means that *those* nodes must run in order to calculate our output, so we add all of them to the stack. Now we look at all of those nodes and see what *their* direct dependencies are and add *those* to the stack. We continue this pattern all the way back in the graph until there are no dependencies left to run, and in this way we guarantee that we have all of the nodes we need to run the graph, and *only* those nodes. In addition, the stack will be ordered in a way that we are guaranteed to be able to run each node in the stack as we iterate through it. The main thing to look out for is to keep track of nodes that were already calculated and to store their value in memory- that way we don't calculate the same node over and over again. By doing this, we are able to make sure our computation is as lean as possible, which can save hours of processing time on huge graphs.

Defining Computation Graphs in TensorFlow

In this book, you're going to be exposed to diverse and fairly complex machine learning models. However, the process of defining each of them in TensorFlow follows a similar pattern. As you dive into various math concepts and learn how to implement them, understanding the core TensorFlow work pattern will keep yourself grounded. Luckily, this workflow is simple to remember- it's only two steps:

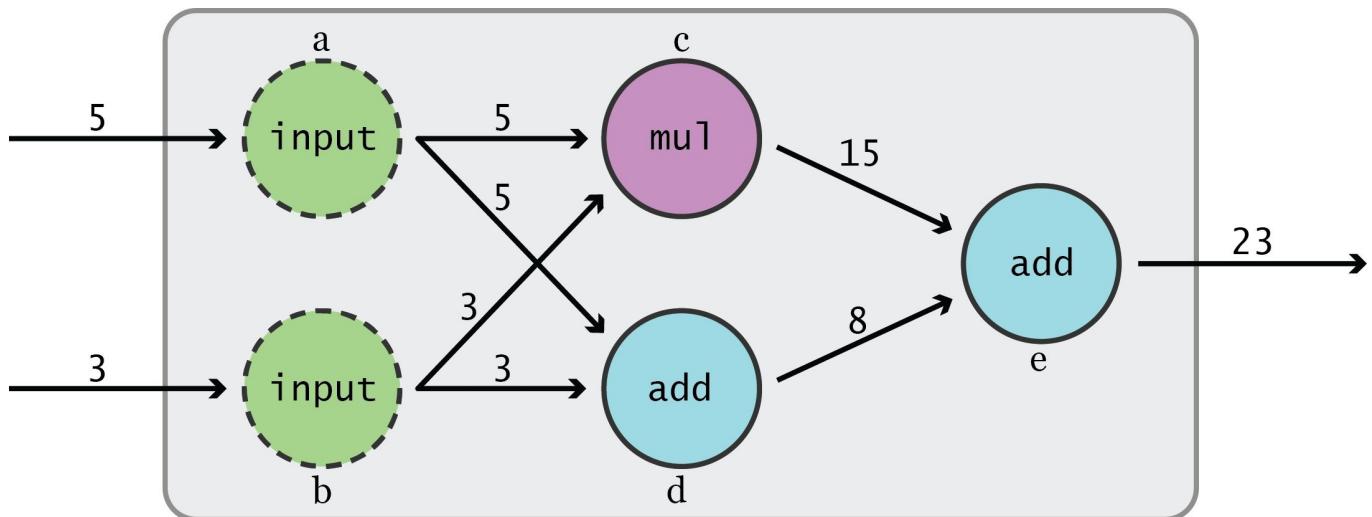
1. *Define* the computation graph
2. *Run* the graph (with data)

This seems obvious- you can't run a graph if it doesn't exist yet! But it's an important distinction to make as the sheer volume of functionality in TensorFlow can be overwhelming when writing your own code. By worrying about only one portion of this workflow at a time, it can help you structure your code more thoughtfully as well as aide in pointing you towards the next thing to work on.

This section will focus on the basics of *defining* graphs in TensorFlow, and the next section will go over *running* a graph once its created. At the end, we'll tie the two together, and show how we can create graphs that change over multiple runs and take in different data.

Building your first TensorFlow graph

We became pretty familiar with the following graph in the last section:



Here's what it looks like in TensorFlow code:

```
import tensorflow as tf

a = tf.constant(5, name="input_a")
b = tf.constant(3, name="input_b")
c = tf.mul(a,b, name="mul_c")
d = tf.add(a,b, name="add_d")
e = tf.add(c,d, name="add_e")
```

Let's break this code down line by line. First, you'll notice this import statement:

```
import tensorflow as tf
```

This, unsurprisingly, imports the TensorFlow library and gives it an alias of `tf`. This is by convention, as it's much easier to type "tf," rather than "tensorflow" over and over as we use its various functions!

Next, let's focus on our first two variable assignments:

```
a = tf.constant(5, name="input_a")
b = tf.constant(3, name="input_b")
```

Here, we're defining our "input" nodes, `a` and `b`. These lines use our first TensorFlow Operation: `tf.constant()`. In TensorFlow, any computation node in the graph is called an **Operation**, or **Op** for short. Ops take in zero or more `Tensor` objects as input and output zero or more `Tensor` objects. To create an Operation, you call its associated Python constructor- in this case, `tf.constant()` creates a "constant" Op. It takes in a single tensor value, and outputs that same value to nodes that are directly connected to it. For convenience, the function automatically converts the scalar numbers 5 and 3 into `Tensor` objects for us. We also pass in an optional string `name` parameter, which we can use to give an identifier to the nodes we create.

Don't worry if you don't fully understand what an Operation or `Tensor` object are at this time, since we'll be going into more detail later in this chapter.

```
c = tf.mul(a,b, name="mul_c")
d = tf.add(a,b, name="add_d")
```

Here, we are defining the next two nodes in our graph, and they both use the nodes we defined previously. Node `c` uses the `tf.mul` Op, which takes in two inputs and outputs the result of multiplying them together. Similarly, node `d` uses `tf.add`, an Operation that outputs the result of adding two inputs together. We again pass in a `name` to both of these Ops (it's something you'll be seeing a lot of). Notice that we don't have to define the edges of the graph separately from the node- when you create a node in TensorFlow, you include all of the inputs that the Operation needs to compute, and the software draws the connections for you.

```
e = tf.add(c,d, name="add_e")
```

This last line defines the final node in our graph. `e` uses `tf.add` in a similar fashion to node `d`. However, this time it takes nodes `c` and `d` as input- exactly as its described in the graph above.

With that, our first, albeit small, graph has been fully defined! If you were to execute the above in a Python script or shell, it would run, but it wouldn't actually do anything. Remember- this is just the *definition* part of the process. To get a brief taste of what running a graph looks like, we could add the following two lines at the end to get our graph to output the final node:

```
sess = tf.Session()  
sess.run(e)
```

If you ran this in an interactive environment, such as the `python` shell or the Jupyter/iPython Notebook, you would see the correct output:

```
...  
>>> sess = tf.Session()  
>>> sess.run(e)  
23
```

That's enough talk for now: let's actually get this running in live code!

Exercise: Building a Basic Graph in TensorFlow

It's time to do it live! In this exercise, you'll code your first TensorFlow graph, run various parts of it, and get your first exposure to the incredibly useful tool **TensorBoard**. When you finish this, you should feel comfortable experimenting with and building basic TensorFlow graphs.

Now, let's actually define it in TensorFlow! Make sure you have TensorFlow installed, and start up your Python dependency environment (Virtualenv, Conda, Docker) if you're using one. In addition, if you installed TensorFlow from source, make sure that your console's present working directory is *not* the TensorFlow source folder, otherwise Python will get confused when we import the library. Now, start an interactive Python session, either using the Jupyter Notebook with the shell command `jupyter notebook`, or start a simple Python shell with `python`. If you have another preferred way of writing Python interactively, feel free to use that!

You could write this as a Python file and run it non-interactively, but the output of running a graph is not displayed by default when doing so. For the sake of seeing the result of your graph, getting immediate feedback on your syntax, and (in the case of the Jupyter Notebook) the ability to fix errors and change code on the fly, we highly recommend doing these examples in an interactive environment. Plus, interactive TensorFlow is fun!

First, we need to load up the TensorFlow library. Write out your import statement as follows:

```
import tensorflow as tf
```

It may think for a few seconds, but afterward it will finish importing and will be ready for the next line of code. If you installed TensorFlow with GPU support, you may see some output notifying you that CUDA libraries were imported. If you get an error that looks like this:

```
ImportError: cannot import name pywrap_tensorflow
```

Make sure that you didn't launch your interactive environment from the TensorFlow source folder. If you get an error that looks like this:

```
ImportError: No module named tensorflow
```

Double check that TensorFlow is installed properly. If you are using Virtualenv or Conda, ensure that your TensorFlow environment is active when you start your interactive Python software. Note that if you have multiple terminals running, one terminal may have an environment active while the other does not.

Assuming the import worked without any hiccups, we can move on to the next portion of the code:

```
a = tf.constant(5, name="input_a")
b = tf.constant(3, name="input_b")
```

This is the same code that we saw above- feel free to change the values or name parameters of these constants. In this book, we'll stick to the same values we had for the sake of consistency.

```
c = tf.mul(a,b, name="mul_c")
d = tf.add(a,b, name="add_d")
```

Next up, we have the first Ops in our code that actually perform a mathematical function. If you're sick and tired of `tf.mul` and `tf.add`, feel free to swap in `tf.sub`, `tf.div`, or `tf.mod`, which perform subtraction, division, or modulo operations, respectively.

`tf.div` performs either integer division or floating point division depending on the type of input provided. If you want to ensure floating point division, try out `tf.truediv`!

Then we can add in our final node:

```
e = tf.add(c,d, name="add_e")
```

You probably noticed that there hasn't been any output when calling these Operations. That's because they have been simply adding Ops to a graph behind the scenes, but no computation is actually taking place. In order to run the graph, we're going to need a TensorFlow Session:

```
sess = tf.Session()
```

Session objects are in charge of supervising graphs as they run, and are the primary interface for running graphs. We're going to discuss Session objects in depth after this exercise, but for now just know that in TensorFlow you need a Session if you want to run your code! We assign our Session to the variable `sess` so we can access it later.

On `InteractiveSession`: There is a slight variation on `tf.Session` called `tf.InteractiveSession`. It's actually designed for use in interactive Python software, such as those you may be using, and it makes a few alternative ways of running code a little simpler. The downsides are that it's less useful for writing TensorFlow in a Python file, and that it abstracts away information that you should learn as a new user to TensorFlow. Besides, in the end it doesn't save *that* many keystrokes. In this book, we'll stick to the standard `tf.Session`.

```
sess.run(e)
```

Here's where we finally can see the result! After running this code, you should see the output of your graph. In our example graph, the output was 23, but it will be different depending the exact functions and inputs you used.

That's not all we can do however. Let's try plugging in one of the other nodes in our graph to `sess.run()`:

```
sess.run(c)
```

You should see the intermediary value of `c` as the output of this call (15, in the example code). TensorFlow doesn't make any assumptions about graphs you create, and for all the program cares node `c` could be the output you want! In fact, you can use the `run()` on any Operation in your graph. When you pass an Op into `sess.run()`, what you are essentially saying to TensorFlow is, "Here is a node I would like to output. Please run all operations necessary to calculate that node". Play around and try outputting some of the other nodes in your graph!

You can also save the output from running the graph- let's save the output from node `e` to a Python variable called `output`:

```
output = sess.run(e)
```

Great! Now that we have a `Session` active and our graph defined, let's visualize it to confirm that it's structured the same way we drew it out. To do that, we're going to use **TensorBoard**, which came installed with TensorFlow. To take advantage of TensorBoard, we're just going to add one line to our code:

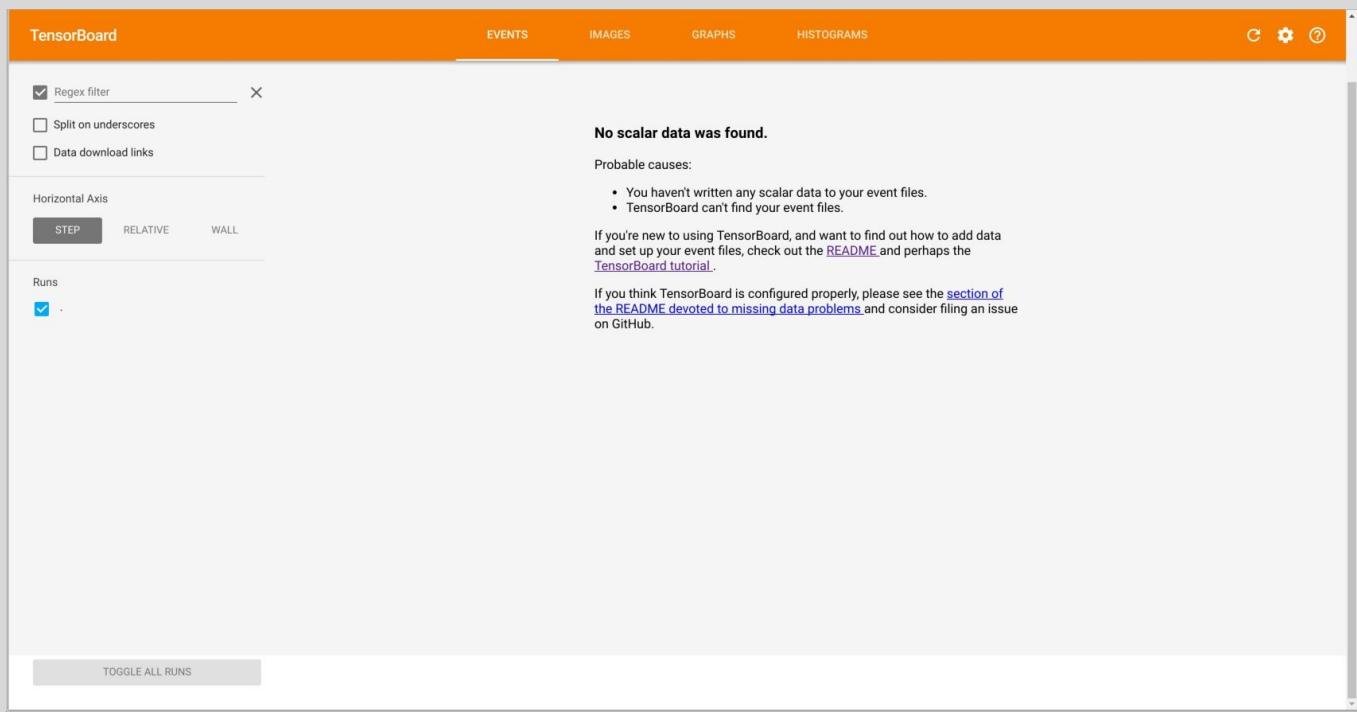
```
writer = tf.train.SummaryWriter('./my_graph', sess.graph)
```

Let's break down what this code does. We are creating a TensorFlow `SummaryWriter` object, and assigning it to the variable `writer`. In this exercise, we won't be performing any additional actions with the `SummaryWriter`, but in the future we'll be using them to save data and summary statistics from our graphs, so we assign it to a variable to get in the habit. We pass in two parameters to initialize `SummaryWriter`. The first is a string output directory, which is where the graph description will be stored on disk. In this case, the files created will be put in a directory called `my_graph`, and will be located inside the directory we are running our Python code. The second input we pass into `SummaryWriter` is the `graph` attribute of our `Session`. `tf.Session` objects, as managers of graphs defined in TensorFlow, have a `graph` attribute that is a reference to the graph they are keeping track of. By passing this on to `SummaryWriter`, the writer will output a description of the graph inside the "my_graph" directory. `SummaryWriter` objects write this data immediately upon initialization, so once you have executed this line of code, we can start up TensorBoard.

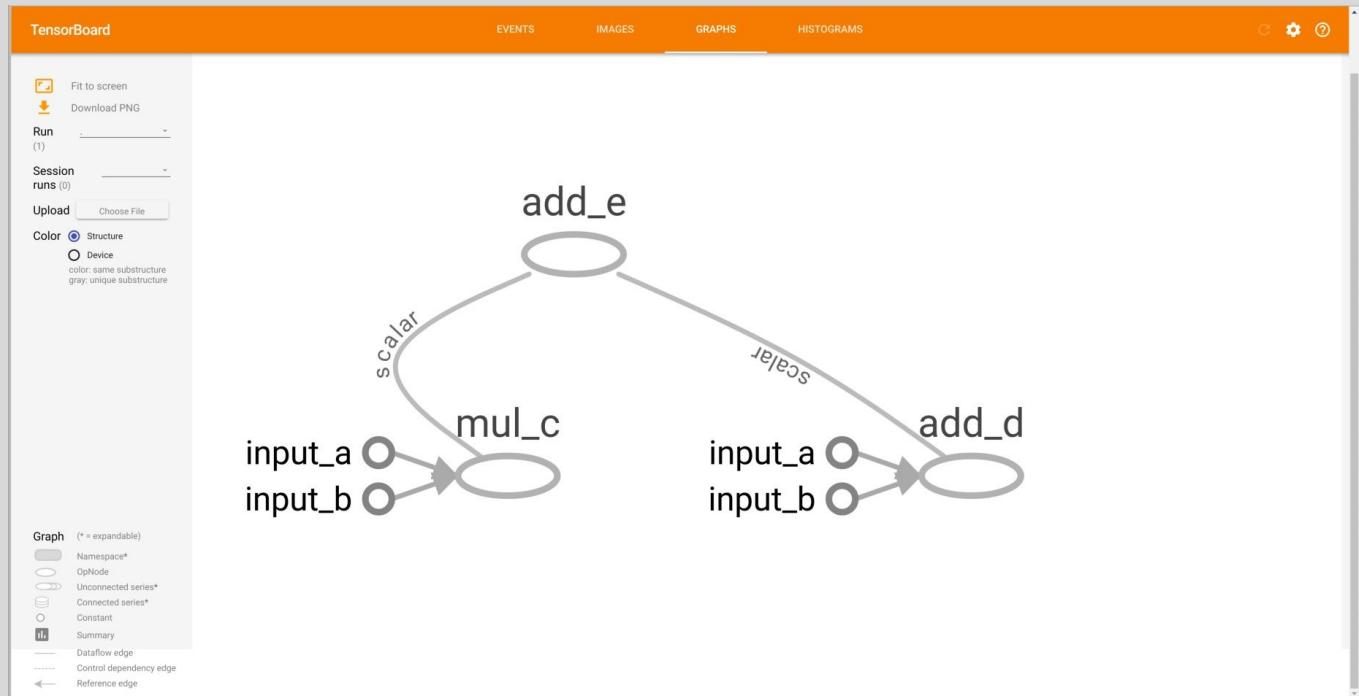
Go to your terminal and type in the following command, making sure that your present working directory is the same as where you ran your Python code (you should see the "my_graph" directory listed):

```
$ tensorboard --logdir="my_graph"
```

You should see some log info print to the console, and then the message "Starting TensorBoard on port 6006". What you've done is start up a TensorBoard server that is using data from the "my_graph" directory. By default, the server started on port 6006- to access TensorBoard, open up a browser and type `http://localhost:6006`. You'll be greeted with an orange-and-white-themed screen:



Don't be alarmed by the "No scalar data was found" warning message. That just means that we didn't save out any summary statistics for TensorBoard to display- normally, this screen would show us information that we asked TensorFlow to save using our `SummaryWriter`. Since we didn't write any additional stats, there's nothing to display. That's fine, though, as we're here to admire our beautiful graph. Click on the "Graphs" link at the top of the page, and you should see a screen similar to this:



That's more like it! If your graph is too small, you can zoom in on TensorBoard by scrolling your mousewheel up. You can see how each of the nodes is labeled based on the name parameter we passed into each `Operation`. If you click on the nodes, you can get information about them such as which other nodes they are attached to. You'll notice that the "inputs", a and b appear to be duplicated, but if you hover or click on either of the nodes labeled "input_a", you should see that they both get a highlighted together. This graph doesn't *look* exactly like the graph we drew above, but it is the same graph since the "input" nodes are simply shown twice. Pretty awesome!

And that's it! You've officially written and run your first ever TensorFlow graph, and you've checked it out in TensorBoard! Not bad for a few lines of code!

For more practice, try adding in a few more nodes, experimenting with some of the different math Ops talked about and adding in a few more `tf.constant` nodes. Run the different nodes you've added and make sure you understand exactly how data is moving through the graph.

Once you are done constructing your graph, let's be tidy and close the `Session` and `SummaryWriter`:

```
writer.close()
sess.close()
```

Technically, `Session` objects close automatically when the program terminates (or, in the interactive case, when you close/restart the Python kernel). However, it's best to explicitly close out of the `Session` to avoid any sort of weird edge case scenarios.

Here's the full Python code after going through this tutorial with our example values:

```
import tensorflow as tf

a = tf.constant(5, name="input_a")
b = tf.constant(3, name="input_b")
c = tf.mul(a,b, name="mul_c")
d = tf.add(a,b, name="add_d")
e = tf.add(c,d, name="add_e")

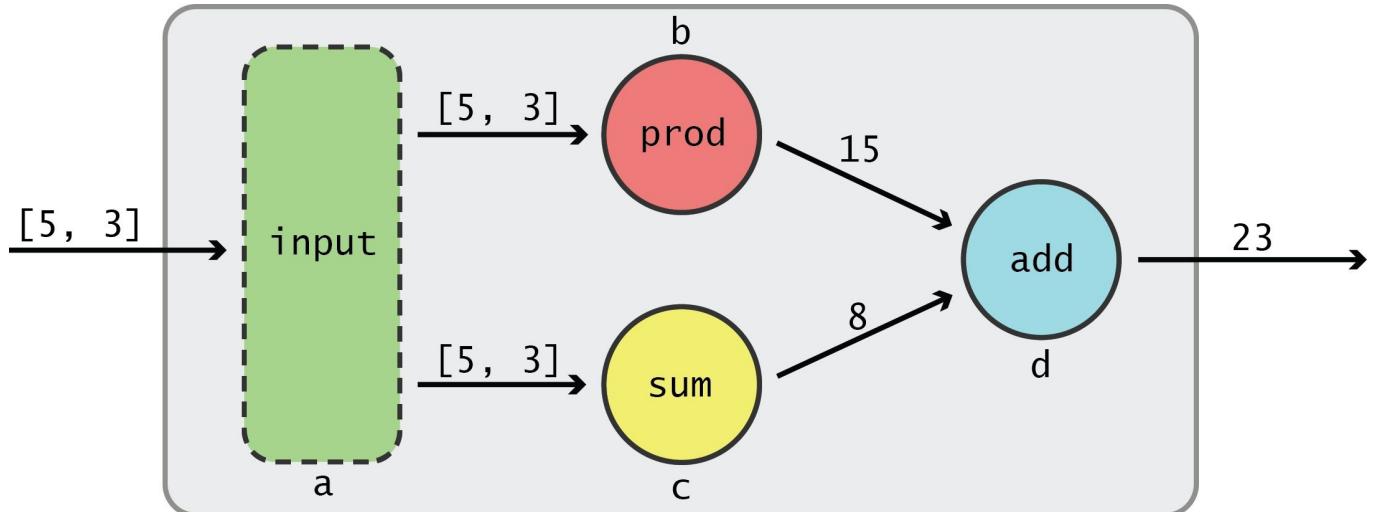
sess = tf.Session()
output = sess.run(e)
writer = tf.train.SummaryWriter('./my_graph', sess.graph)

writer.close()
sess.close()
```


Thinking with tensors

Simple, scalar numbers are great when learning the basics of computation graphs, but now that we have a grasp of the “flow”, let’s get acquainted with tensors.

Tensors, as mentioned before, are simply the n-dimensional abstraction of matrices. So a 1-D tensor would be equivalent to a vector, a 2-D tensor is a matrix, and above that you can just say “N-D tensor”. With this in mind, we can modify our previous example graph to use tensors:



Now, instead of having two separate input nodes, we have a single node that can take in a vector (or 1-D tensor) of numbers. This graph has several advantages over our previous example:

1. The client only has to send input to a single node, which simplifies using the graph.
2. The nodes that directly depend on the input now only have to keep track of one dependency instead of two.
3. We now have the option of making the graph take in vectors of any length, if we’d like. This would make the graph more flexible. We can also have the graph enforce a strict requirement, and force inputs to be of length two (or any length we’d like)

We can implement this change in TensorFlow by modifying our previous code:

```
import tensorflow as tf

a = tf.constant([5,3], name="input_a")
b = tf.reduce_prod(a, name="prod_b")
c = tf.reduce_sum(a, name="sum_c")
d = tf.add(b,c, name="add_d")
```

Aside from adjusting the variable names, we made two main changes here:

1. We replaced the separate nodes `a` and `b` with a consolidated input node (now just `a`). We passed in a list of numbers, which `tf.constant` is able to convert to a 1-D `Tensor`
2. Our multiplication and addition Operations, which used to take in scalar values, are now `tf.reduce_prod()` and `tf.reduce_sum()`. These functions, when just given a `Tensor` as input, take all of its values and either multiply or sum them up, respectively.

In TensorFlow, all data passed from node to node are `Tensor` objects. As we’ve seen,

TensorFlow Operations are able to look at standard Python types, such as integers and strings, and automatically convert them into tensors. There are a variety of ways to create `Tensor` objects manually (that is, without reading it in from an external data source), so let's go over a few of them.

In this book, when discussing code we will use “tensor” and “Tensor” interchangeably.

Python Native Types

TensorFlow can take in Python numbers, booleans, strings, or lists of any of the above. Single values will be converted to a 0-D `Tensor` (or scalar), lists of values will be converted to a 1-D `Tensor` (vector), lists of lists of values will be converted to a 2-D `Tensor` (matrix), and so on. Here's a small chart showcasing this:

```
t_0 = 50                                # Treated as 0-D Tensor, or "scalar"  
t_1 = [b"apple", b"peach", b"grape"] # Treated as 1-D Tensor, or "vector"  
t_2 = [[True, False, False],           # Treated as 2-D Tensor, or "matrix"  
       [False, False, True],  
       [False, True, False]]  
t_3 = [[[0, 0], [0, 1], [0, 2]],      # Treated as 3-D Tensor  
       [[1, 0], [1, 1], [1, 2]],  
       [[2, 0], [2, 1], [2, 2]]]  
...  
...
```

TensorFlow data types

We haven't seen booleans or strings yet, but you can think of tensors as a way to store any data in a structured format. Obviously, math functions don't work on strings, and string-parsing functions don't work on numbers, but it's good to know that TensorFlow can handle more than just numerics! Here's the [full list of data types available in TensorFlow](#):

Data type (dtype) Description

<code>tf.float32</code>	32-bit floating point
<code>tf.float64</code>	64-bit floating point
<code>tf.int8</code>	8-bit signed integer
<code>tf.int16</code>	16-bit signed integer
<code>tf.int32</code>	32-bit signed integer
<code>tf.int64</code>	64-bit signed integer
<code>tf.uint8</code>	8-bit unsigned integer
<code>tf.string</code>	String (as bytes array, <i>not</i> Unicode)

<code>tf.bool</code>	Boolean
<code>tf.complex64</code>	Complex number, with 32-bit floating point real portion, and 32-bit floating point imaginary portion
<code>tf.qint8</code>	8-bit signed integer (used in quantized Operations)
<code>tf.qint32</code>	32-bit signed integer (used in quantized Operations)
<code>tf.quint8</code>	8-bit unsigned integer (used in quantized Operations)

Using Python types to specify `Tensor` objects is quick and easy, and it is useful for prototyping ideas. However, there is an important and unfortunate downside to doing it this way. TensorFlow has a plethora of data types at its disposal, but basic Python types lack the ability to explicitly state what kind of data type you'd like to use. Instead, TensorFlow has to infer which data type you meant. With some types, such as strings, this is simple, but for others it may be impossible. For example, in Python all integers [are the same type](#), but TensorFlow has 8-bit, 16-bit, 32-bit, and 64-bit integers available. There are ways to convert the data into the appropriate type when you pass it into TensorFlow, but certain data types still may be difficult to declare correctly, such as complex numbers. Because of this, it is common to see hand-defined `Tensor` objects as [NumPy](#) arrays.

NumPy arrays

TensorFlow is tightly integrated with [NumPy](#), the scientific computing package designed for manipulating N-dimensional arrays. If you don't have experience with NumPy, we highly recommend looking at the wealth of tutorials and documentation available for the library, as it has become part of the *lingua franca* of data science. TensorFlow's data types are based on [those from NumPy](#); in fact, the statement `np.int32 == tf.int32` returns `True`! Any NumPy array can be passed into any TensorFlow Op, and the beauty is that you can easily specify the data type you need with minimal effort.

STRING DATA TYPES

There is a “gotcha” here for string data types. For numeric and boolean types, TensorFlow and NumPy dtypes match down the line. However, `tf.string` does not have an exact match in NumPy due to the way NumPy handles strings. That said, TensorFlow can import string arrays from NumPy perfectly fine- just don’t specify a `dtype` in NumPy!

As a bonus, you can use the functionality of the `numpy` library both before and after running your graph, as the tensors returned from `session.run` *are* NumPy arrays. Here’s an example of how to create NumPy arrays, mirroring the above example.

```
import numpy as np # Don't forget to import NumPy!

# 0-D Tensor with 32-bit integer data type
t_0 = np.array(50, dtype=np.int32)

# 1-D Tensor with byte string data type
# Note: don't explicitly specify dtype when using strings in NumPy
t_1 = np.array([b"apple", b"peach", b"grape"])

# 1-D Tensor with boolean data type
t_2 = np.array([[True, False, False],
                [False, False, True],
                [False, True, False]],
                dtype=np.bool)

# 3-D Tensor with 64-bit integer data type
t_3 = np.array([[[0, 0], [0, 1], [0, 2]],
                [[1, 0], [1, 1], [1, 2]],
                [[2, 0], [2, 1], [2, 2]]],
                dtype=np.int64)
...
```

Although TensorFlow is designed to understand NumPy data types natively, the converse is not true. Don’t accidentally try to initialize a NumPy array with `tf.int32`!

[FOOTNOTE] Technically, NumPy is able to automatically detect data types as well, but it really is best to start getting in the habit of being explicit about the numeric properties you want your `Tensor` objects to have. When you’re dealing with huge graphs, you *really* don’t want to have to hunt down which objects are causing a `TypeMismatchError`! The one exception to this is when dealing with strings- don’t bother specifying a `dtype` when creating a string `Tensor`.

Using NumPy is the recommended way of specifying `Tensor` objects by hand!

Tensor shape

Throughout the TensorFlow library, you'll commonly see functions and Operations that refer to a tensor's "shape". The shape, in TensorFlow terminology, describes both the number dimensions in a tensor as well as the length of each dimension. Tensor shapes can either be Python lists or tuples containing an ordered set of integers: there are as many numbers in the list as there are dimensions, and each number describes the length of its corresponding dimension. For example, the list [2, 3] describes the shape of a 2-D tensor of length 2 in its first dimension and length 3 in its second dimension. Note that either tuples (wrapped with parentheses ()) or lists (wrapped with brackets []) can be used to define shapes. Let's take a look at more examples to illustrate this further:

```
# Shapes that specify a 0-D Tensor (scalar)
# e.g. any single number: 7, 1, 3, 4, etc.
s_0_list = []
s_0_tuple = ()

# Shape that describes a vector of length 3
# e.g. [1, 2, 3]
s_1 = [3]

# Shape that describes a 3-by-2 matrix
# e.g. [[1, 2],
#        [3, 4],
#        [5, 6]]
s_2 = (3, 2)
```

In addition to being able to specify fixed lengths to each dimension, you are also able to assign a flexible length by passing in `None` as a dimension's value. Furthermore, passing in the value `None` as a shape (instead of using a list/tuple that contains `None`), will tell TensorFlow to allow a tensor of any shape. That is, a tensor with any amount of dimensions and any length for each dimension:

```
# Shape for a vector of any length:
s_1_flex = [None]

# Shape for a matrix that is any amount of rows tall, and 3 columns wide:
s_2_flex = (None, 3)

# Shape of a 3-D Tensor with length 2 in its first dimension, and variable-
# length in its second and third dimensions:
s_3_flex = [2, None, None]

# Shape that could be any Tensor
s_any = None
```

If you ever need to figure out the shape of a tensor in the middle of your graph, you can use the `tf.shape` Op. It simply takes in the Tensor object you'd like to find the shape for, and returns it as an `int32` vector:

```
import tensorflow as tf

# ...create some sort of mystery tensor

# Find the shape of the mystery tensor
shape = tf.shape(mystery_tensor, name="mystery_shape")
```

Remember that `tf.shape`, like any other Operation, doesn't run until it is executed inside of a Session.

REMINDER!

Tensors are just a superset of matrices!

TensorFlow operations

As mentioned earlier, [TensorFlow Operations](#), also known as **Ops**, are nodes that perform computations on or with Tensor objects. After computation, they return zero or more tensors, which can be used by other Ops later in the graph. To create an Operation, you call its constructor in Python, which takes in whatever Tensor parameters needed for its calculation, known as *inputs*, as well as any additional information needed to properly create the Op, known as *attributes*. The Python constructor returns a handle to the Operation's *output* (zero or more `Tensor` objects), and it is this output which can be passed on to other Operations or `Session.run`:

```
import tensorflow as tf
import numpy as np

# Initialize some tensors to use in computation
a = np.array([2, 3], dtype=np.int32)
b = np.array([4, 5], dtype=np.int32)

# Use `tf.add()` to initialize an "add" Operation
# The variable `c` will be a handle to the Tensor output of this Op
c = tf.add(a, b)
```

ZERO-INPUT, ZERO-OUTPUT OPERATIONS

Yes, that means there are Ops that technically take in zero inputs and return zero outputs. Ops are more than just mathematical computations, and are used for tasks such as initializing state. We'll be going over some of these non-mathematical Operations in this chapter, but for now just remember that not all nodes need to be connected to other nodes.

In addition to *inputs* and *attributes*, each Operation constructor accepts a string parameter, `name`, as input. As we saw in the exercise above, providing a `name` allows us to refer to a specific Op by a descriptive string:

```
c = tf.add(a, b, name="my_add_op")
```

In this example, we give the name “my_add_op” to the add Operation, which we’ll be able to refer to when using tools such as TensorBoard.

You may find that you’ll want to reuse the same name for different Operations in a graph. Instead of manually adding prefixes or suffixes to each name, you can use a `name_scope` to group operations together programmatically. We’ll go over the basic use of name scopes in the exercise at the end of this chapter.

Overloaded operators

TensorFlow also overloads common mathematical operators to make multiplication, addition, subtraction, and other common operations more concise. If one or more arguments to the operator is a `Tensor` object, a TensorFlow Operation will be called and added to the graph. For example, you can easily add two tensors together like this:

```
# Assume that `a` and `b` are `Tensor` objects with matching shapes
c = a + b
```

Here is a complete list of overloaded operators for tensors:

Unary operators

Operator	Related TensorFlow Operation	Description
<code>-x</code>	tf.neg()	Returns the negative value of each element in x
<code>~x</code>	tf.logical_not()	Returns the logical NOT of each element in x. Only compatible with <code>Tensor</code> objects with <code>dtype</code> of <code>tf.bool</code>
<code>abs(x)</code>	tf.abs()	Returns the absolute value of each element in x

Binary operators

Operator	Related TensorFlow Operation	Description
<code>x + y</code>	tf.add()	Add x and y, element-wise

<code>x - y</code>	<code>tf.sub()</code>	Subtract y from x, element-wise
<code>x * y</code>	<code>tf.mul()</code>	Multiply x and y, element-wise
<code>x / y</code> (Python 2)	<code>tf.div()</code>	Will perform element-wise integer division when given an integer type tensor, and floating point (“true”) division on floating point tensors
<code>x / y</code> (Python 3)	<code>tf.truediv()</code>	Element-wise floating point division (including on integers)
<code>x // y</code> (Python 3)	<code>tf.floordiv()</code>	Element-wise floor division, not returning any remainder from the computation
<code>x % y</code>	<code>tf.mod()</code>	Element-wise modulo
<code>x ** y</code>	<code>tf.pow()</code>	The result of raising each element in x to its corresponding element y, element-wise
<code>x < y</code>	<code>tf.less()</code>	Returns the truth table of $x < y$, element-wise
<code>x <= y</code>	<code>tf.less_equal()</code>	Returns the truth table of $x \leq y$, element-wise
<code>x > y</code>	<code>tf.greater()</code>	Returns the truth table of $x > y$, element-wise
<code>x >= y</code>	<code>tf.greater_equal()</code>	Returns the truth table of $x \geq y$, element-wise
<code>x & y</code>	<code>tf.logical_and()</code>	Returns the truth table of $x \& y$, element-wise. <code>dtype</code> must be <code>tf.bool</code>
<code>x y</code>	<code>tf.logical_or()</code>	Returns the truth table of $x y$, element-wise. <code>dtype</code> must be <code>tf.bool</code>
<code>x ^ y</code>	<code>tf.logical_xor()</code>	Returns the truth table of $x ^ y$, element-wise. <code>dtype</code> must be <code>tf.bool</code>

Using these overloaded operators can be great when quickly putting together code, but you will not be able to give name values to each of these Operations. If you need to pass in a name to the Op, call the TensorFlow Operation directly.

Technically, the `==` operator is overloaded as well, but it will not return a `Tensor` of boolean values. Instead, it will return `True` if the two tensors being compared are the same object, and `False` otherwise. This is mainly used for internal purposes. If you’d like to check for equality or inequality, check out [`tf.equal\(\)`](#) and [`tf.not_equal`](#), respectively.

TensorFlow graphs

Thus far, we've only referenced "the graph" as some sort of abstract, omni-presence in TensorFlow, and we haven't questioned how Operations are automatically attached to a graph when we start coding. Now that we've seen some examples, let's take a look at the [TensorFlow Graph object](#), learn how to create more of them, use multiple graphs in conjunction with one another.

Creating a `Graph` is simple- its constructor doesn't take any variables:

```
import tensorflow as tf

# Create a new graph:
g = tf.Graph()
```

Once we have our `Graph` initialized, we can add Operations to it by using the `Graph.as_default()` method to access its context manager. In conjunction with the `with` statement, we can use the context manager to let TensorFlow know that we want to add Operations to a specific `Graph`:

```
with g.as_default():
    # Create Operations as usual; they will be added to graph `g`
    a = tf.mul(2, 3)
    ...
```

You might be wondering why we haven't needed to specify the graph we'd like to add our Ops to in the previous examples. As a convenience, TensorFlow automatically creates a `Graph` when the library is loaded and assigns it to be the default. Thus, any Operations, tensors, etc. defined outside of a `Graph.as_default()` context manager will automatically be placed in the default graph:

```
# Placed in the default graph
in_default_graph = tf.add(1, 2)

# Placed in graph `g`
with g.as_default():
    in_graph_g = tf.mul(2, 3)

# We are no longer in the `with` block, so this is placed in the default graph
also_in_default_graph = tf.sub(5, 1)
```

If you'd like to get a handle to the default graph, use the [`tf.get_default_graph\(\)`](#) function:

```
default_graph = tf.get_default_graph()
```

In most TensorFlow programs, you will only ever deal with the default graph. However, creating multiple graphs can be useful if you are defining multiple models that do not have interdependencies. When defining multiple graphs in one file, it's best practice to either not use the default graph or immediately assign a handle to it. This ensures that nodes are added to each graph in a uniform manner:

Correct - Create new graphs, ignore default graph:

```
import tensorflow as tf

g1 = tf.Graph()
g2 = tf.Graph()

with g1.as_default():
    # Define g1 Operations, tensors, etc.
    ...
```

```
with g2.as_default():
    # Define g2 Operations, tensors, etc.
    ...
```

Correct - Get handle to default graph

```
import tensorflow as tf

g1 = tf.get_default_graph()
g2 = tf.Graph()

with g1.as_default():
    # Define g1 Operations, tensors, etc.
    ...

with g2.as_default():
    # Define g2 Operations, tensors, etc.
    ...
```

Incorrect: Mix default graph and user-created graph styles

```
import tensorflow as tf

g2 = tf.Graph()

# Define default graph Operations, tensors, etc.
...

with g2.as_default():
    # Define g2 Operations, tensors, etc.
    ...
```

Additionally, it is possible to load in previously defined models from other TensorFlow scripts and assign them to `Graph` objects using a combination of the [`Graph.as_graph_def\(\)`](#) and [`tf.import_graph_def`](#) functions. Thus, a user can compute and use the output of several separate models in the same Python file. We will cover importing and exporting graphs later in this book.

TensorFlow Sessions

Sessions, as discussed in the previous exercise, are responsible for graph execution. The constructor

https://www.tensorflow.org/versions/master/api_docs/python/client.html#Session.init [tf.Session.init]

takes in three optional parameters:

- `target` specifies the execution engine to use. For most applications, this will be left at its default empty string value. When using sessions in a distributed setting, this parameter is used to connect to `tf.train.Server` instances (covered in the later chapters of this book).
- `graph` specifies the `Graph` object that will be launched in the `session`. The default value is `None`, which indicates that the current default graph should be used. When using multiple graphs, it's best to explicitly pass in the `Graph` you'd like to run (instead of creating the `session` inside of a `with` block).
- `config` allows users to specify options to configure the session, such as limiting the number of CPUs or GPUs to use, setting optimization parameters for graphs, and logging options.

In a typical TensorFlow program, `session` objects will be created without changing any of the default construction parameters.

```
import tensorflow as tf

# Create Operations, Tensors, etc (using the default graph)
a = tf.add(2, 5)
b = tf.mul(a, 3)

# Start up a `Session` using the default graph
sess = tf.Session()
```

Note that these two calls are identical:

```
sess = tf.Session()
sess = tf.Session(graph=tf.get_default_graph())
```

Once a `session` is opened, you can use its primary method, `run()`, to calculate the value of a desired `Tensor` output:

```
sess.run(b) # Returns 21
```

`Session.run()` takes in one required parameter, `fetches`, as well as three optional parameters: `feed_dict`, `options`, and `run_metadata`. We won't cover `options` or `run_metadata`, as they are still experimental (thus prone to being changed) and are of limited use at this time. `feed_dict`, however, is important to understand and will be covered below.

Fetches

`fetches` accepts any graph element (either an `Operation` or `Tensor` object), which specifies what the user would like to execute. If the requested object is a `Tensor`, then the output of `run()` will be a NumPy array. If the object is an `Operation`, then the output will be `None`.

In the above example, we set `fetches` to the tensor `b` (the output of the `tf.mul` Operation). This tells TensorFlow that the Session should find all of the nodes necessary to compute

the value of `b`, execute them in order, and output the value of `b`. We can also pass in a list of graph elements:

```
sess.run([a, b]) # returns [7, 21]
```

When `fetches` is a list, the output of `run()` will be a list with values corresponding to the output of the requested elements. In this example, we ask for the values of `a` and `b`, in that order. Since both `a` and `b` are tensors, we receive their values as output.

In addition using `fetches` to get `Tensor` outputs, you'll also see examples where we give `fetches` a direct handle to an `operation` which is a useful side-effect when run. An example of this is `tf.initialize_all_variables()`, which prepares all TensorFlow `variable` objects to be used (`variable` objects will be covered later in this chapter). We still pass the `Op` as the `fetches` parameter, but the result of `Session.run()` will be `None`:

```
# Performs the computations needed to initialize Variables, but returns `None`
sess.run(tf.initialize_all_variables())
```

Feed dictionary

The parameter `feed_dict` is used to override `Tensor` values in the graph, and it expects a Python dictionary object as input. The keys in the dictionary are handles to `Tensor` objects that should be overridden, while the values can be numbers, strings, lists, or NumPy arrays (as described previously). The values must be of the same type (or able to be converted to the same type) as the `Tensor` key. Let's show how we can use `feed_dict` to overwrite the value of `a` in the previous graph:

```
import tensorflow as tf

# Create Operations, Tensors, etc (using the default graph)
a = tf.add(2, 5)
b = tf.mul(a, 3)

# Start up a `Session` using the default graph
sess = tf.Session()

# Define a dictionary that says to replace the value of `a` with 15
replace_dict = {a: 15}

# Run the session, passing in `replace_dict` as the value to `feed_dict`
sess.run(b, feed_dict=replace_dict) # returns 45
```

Notice that even though `a` would normally evaluate to 7, the dictionary we passed into `feed_dict` replaced that value with 15. `feed_dict` can be extremely useful in a number of situations. Because the value of a tensor is provided up front, the graph no longer needs to compute any of the tensor's normal dependencies. This means that if you have a large graph and want to test out part of it with dummy values, TensorFlow won't waste time with unnecessary computations. `feed_dict` is also useful for specifying input values, as we'll cover in the upcoming placeholder section.

After you are finished using the `Session`, call its `close()` method to release unneeded resources:

```
# Open Session
sess = tf.Session()

# Run the graph, write summary statistics, etc.
...
```

```
# Close the graph, release its resources
sess.close()
```

As an alternative, you can also use the `Session` as a context manager, which will automatically close when the code exits its scope:

```
with tf.Session() as sess:
    # Run graph, write summary statistics, etc.
    ...
# The Session closes automatically
```

We can also use a `Session` as a context manager by using its [`as_default\(\)`](#) method. Similarly to how `Graph` objects can be used implicitly by certain Operations, you can set a session to be used automatically by certain functions. The most common of such functions are [`Operation.run\(\)`](#) and [`Tensor.eval\(\)`](#), which act as if you had passed them in to `Session.run()` directly.

```
# Define simple constant
a = tf.constant(5)

# Open up a Session
sess = tf.Session()

# Use the Session as a default inside of `with` block
with sess.as_default():
    a.eval()

# Have to close Session manually.
sess.close()
```

MORE ON INTERACTIVESession

Earlier in the book, we mentioned that `InteractiveSession` is another type of TensorFlow session, but that we wouldn't be using it. All `InteractiveSession` does is automatically make itself the default session in the runtime. This can be handy when using an interactive Python shell, as you can use `a.eval()` or `a.run()` instead of having to explicitly type out `sess.run([a])`. However, if you need to juggle multiple sessions, things can get a little tricky. We find that maintaining a consistent way of running graphs makes debugging much easier, so we're sticking with regular `Session` objects.

Now that we've got a firm understanding of running our graph, let's look at how to properly specify input nodes and use `feed_dict` in conjunction with them.

Adding Inputs with Placeholder nodes

You may have noticed that the graph we defined previously doesn't use true "input"; it always uses the same numbers, 5 and 3. What we would like to do instead is take values from the client so that we can reuse the transformation described by our graph with all sorts of different numbers. We do that with what is called a "placeholder". Placeholders, as the name implies, act as if they are Tensor objects, but they do not have their values specified when created. Instead, they hold the place for a Tensor that will be fed at runtime, in effect becoming an "input" node. Creating placeholders is done using the [tf.placeholder](#) Operation:

```
import tensorflow as tf
import numpy as np

# Creates a placeholder vector of length 2 with data type int32
a = tf.placeholder(tf.int32, shape=[2], name="my_input")

# Use the placeholder as if it were any other Tensor object
b = tf.reduce_prod(a, name="prod_b")
c = tf.reduce_sum(a, name="sum_c")

# Finish off the graph
d = tf.add(b, c, name="add_d")
```

`tf.placeholder` takes in a required parameter `dtype`, as well as the optional parameter `shape`:

- `dtype` specifies the data type of values that will be passed into the placeholder. This is required, as it is needed to ensure that there will be no type mismatch errors.
- `shape` specifies what shape the fed Tensor will be. See the discussion on Tensor shapes above. The default value of `shape` is `None`, which means a Tensor of any shape will be accepted.

Like any Operation, you can also specify a `name` identifier to `tf.placeholder`.

In order to actually give a value to the placeholder, we'll use the `feed_dict` parameter in `session.run()`. We use the handle to the placeholder's output as the key to the dictionary (in the above code, the variable `a`), and the Tensor object we want to pass in as its value:

```
# Open a TensorFlow Session
sess = tf.Session()

# Create a dictionary to pass into `feed_dict`
# Key: `a`, the handle to the placeholder's output Tensor
# Value: A vector with value [5, 3] and int32 data type
input_dict = {a: np.array([5, 3], dtype=np.int32)}

# Fetch the value of `d`, feeding the values of `input_vector` into `a`
sess.run(d, feed_dict=input_dict)
```

You *must* include a key-value pair in `feed_dict` for each placeholder that is a dependency of the fetched output. Above, we fetched `d`, which depends on the output of `a`. If we had defined additional placeholders that `d` did not depend on, we would not need to include them in the `feed_dict`.

You cannot fetch the value of placeholders- it will simply raise an exception if you try to feed one into `Session.run()`.

Variables

Creating variables

Tensor and Operation objects are immutable, but machine learning tasks, by their nature, need a mechanism to save changing values over time. This is accomplished in TensorFlow with [variable](#) objects, which contain mutable tensor values that persist across multiple calls to `Session.run()`. You can create a `variable` by using its constructor, `tf.Variable()`:

```
import tensorflow as tf

# Pass in a starting value of three for the variable
my_var = tf.Variable(3, name="my_variable")
```

Variables can be used in TensorFlow functions/Operations anywhere you might use a `Tensor`; its present value will be passed on to the Operation using it:

```
add = tf.add(5, my_var)
mul = tf.mul(8, my_var)
```

The initial value of Variables will often be large tensors of zeros, ones, or random values. To make it easier to create these common values, TensorFlow has a number of helper Ops, such as [tf.zeros\(\)](#), [tf.ones\(\)](#), [tf.random_normal\(\)](#), and [tf.random_uniform\(\)](#), each of which takes in a `shape` parameter which specifies the dimension of the desired `Tensor`:

```
# 2x2 matrix of zeros
zeros = tf.zeros([2, 2])

# vector of length 6 of ones
ones = tf.ones([6])

# 3x3x3 Tensor of random uniform values between 0 and 10
uniform = tf.random_uniform([3, 3, 3], minval=0, maxval=10)

# 3x3x3 Tensor of normally distributed numbers; mean 0 and standard deviation 2
normal = tf.random_normal([3, 3, 3], mean=0.0, stddev=2.0)
```

Instead of using `tf.random_normal()`, you'll often see use of [tf.truncated_normal\(\)](#) instead, as it doesn't create any values more than two standard deviations away from its mean. This prevents the possibility of having one or two numbers be significantly different than the other values in the tensor:

```
# No values below 3.0 or above 7.0 will be returned in this Tensor
trunc = tf.truncated_normal([2, 2], mean=5.0, stddev=1.0)
```

You can pass in these Operations as the initial values of Variables as you would a hand-written `Tensor`:

```
# Default value of mean=0.0
# Default value of stddev=1.0
random_var = tf.Variable(tf.truncated_normal([2, 2]))
```

Variable Initialization

`Variable` objects live in the `Graph` like most other TensorFlow objects, but their state is actually managed by a `Session`. Because of this, Variables have an extra step involved in order to use them- you must *initialize* the `variable` within a `session`. This causes the `Session` to start keeping track of the ongoing value of the `variable`. This is typically done by passing in the [tf.initialize_all_variables\(\)](#) Operation to `session.run()`:

```
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
```

If you'd only like to initialize a subset of Variables defined in the graph, you can use `tf.initialize_variables()`, which takes in a list of Variables to be initialized:

```
var1 = tf.Variable(0, name="initialize_me")
var2 = tf.Variable(1, name="no_initialization")
init = tf.initialize_variables([var1], name="init_var1")
sess = tf.Session()
sess.run(init)
```

Changing Variables

In order to change the value of the variable, you can use the `Variable.assign()` method, which gives the variable the new value to be. Note that `variable.assign()` is an Operation, and must be run in a Session to take effect:

```
# Create variable with starting value of 1
my_var = tf.Variable(1)

# Create an operation that multiplies the variable by 2 each time it is run
my_var_times_two = my_var.assign(my_var * 2)

# Initialization operation
init = tf.initialize_all_variables()

# Start a session
sess = tf.Session()

# Initialize variable
sess.run(init)

# Multiply variable by two and return it
sess.run(my_var_times_two)
## OUT: 2

# Multiply again
sess.run(my_var_times_two)
## OUT: 4

# Multiply again
sess.run(my_var_times_two)
## OUT: 8
```

For simple incrementing and decrementing of Variables, TensorFlow includes the `Variable.assign_add()` `Variable.assign_sub()` methods:

```
# Increment by 1
sess.run(my_var.assign_add(1))

# Decrement by 1
sess.run(my_var.assign_sub(1))
```

Because Sessions maintain variable values separately, each session can have its own current value for a variable defined in a graph:

```
# Create Ops
my_var = tf.Variable(0)
init = tf.initialize_all_variables()

# Start Sessions
sess1 = tf.Session()
sess2 = tf.Session()

# Initialize Variable in sess1, and increment value of my_var in that Session
sess1.run(init)
sess1.run(my_var.assign_add(5))
## OUT: 5
```

```

# Do the same with sess2, but use a different increment value
sess2.run(init)
sess2.run(my_var.assign_add(2))
## OUT: 2

# Can increment the Variable values in each Session independently

sess1.run(my_var.assign_add(5))
## OUT: 10

sess2.run(my_var.assign_add(2))
## OUT: 4

```

If you'd like to reset your Variables to their starting value, simply call `tf.initialize_all_variables()` again (or `tf.initialize_variables` if you only want to reset a subset of them):

```

# Create Ops
my_var = tf.Variable(0)
init = tf.initialize_all_variables()

# Start Session
sess = tf.Session()

# Initialize Variables
sess.run(init)

# Change the Variable
sess.run(my_var.assign(10))

# Reset the Variable to 0, its initial value
sess.run(init)

```

Trainable

Later in this book, you'll see various `optimizer` classes which automatically train machine learning models. That means that it will change values of `variable` objects without explicitly asking to do so. In most cases, this is what you want, but if there are Variables in your graph that should *only* be changed manually and not with an `optimizer`, you need to set their `trainable` parameter to `False` when creating them:

```
not_trainable = tf.Variable(0, trainable=False)
```

This is typically done with step counters or anything else that isn't going to be involved in the calculation of a machine learning model.

Organizing your graph with name scopes

We've now covered the core building blocks necessary to build any TensorFlow graph. So far, we've only worked with toy graphs containing a few nodes and small tensors, but real world models can contain dozens or hundreds of nodes, as well as millions of parameters. In order to manage this level of complexity, TensorFlow currently offers a mechanism to help organize your graphs: *name scopes*.

Name scopes are incredibly simple to use and provide great value when visualizing your graph with TensorBoard. Essentially, name scopes allow you to group Operations into larger, named blocks. Then, when you launch your graph with TensorBoard, each name scope will encapsulate its own Ops, making the visualization much more digestible. For basic name scope usage, simply add your Operations in a `with tf.name_scope(<name>)` block:

```
import tensorflow as tf

with tf.name_scope("Scope_A"):
    a = tf.add(1, 2, name="A_add")
    b = tf.mul(a, 3, name="A_mul")

with tf.name_scope("Scope_B"):
    c = tf.add(4, 5, name="B_add")
    d = tf.mul(c, 6, name="B_mul")

e = tf.add(b, d, name="output")
```

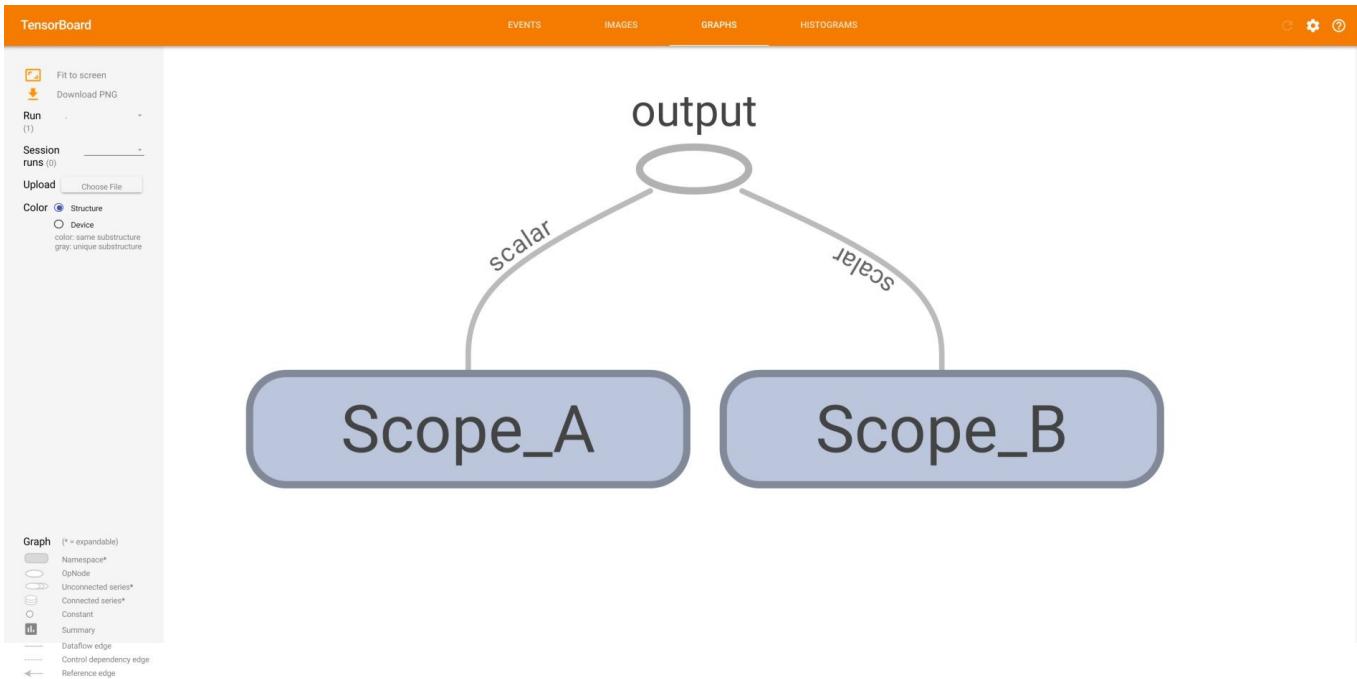
To see the result of these name scopes in TensorBoard, let's open up a `summarywriter` and write this graph to disk.

```
writer = tf.train.SummaryWriter('./name_scope_1', graph=tf.get_default_graph())
writer.close()
```

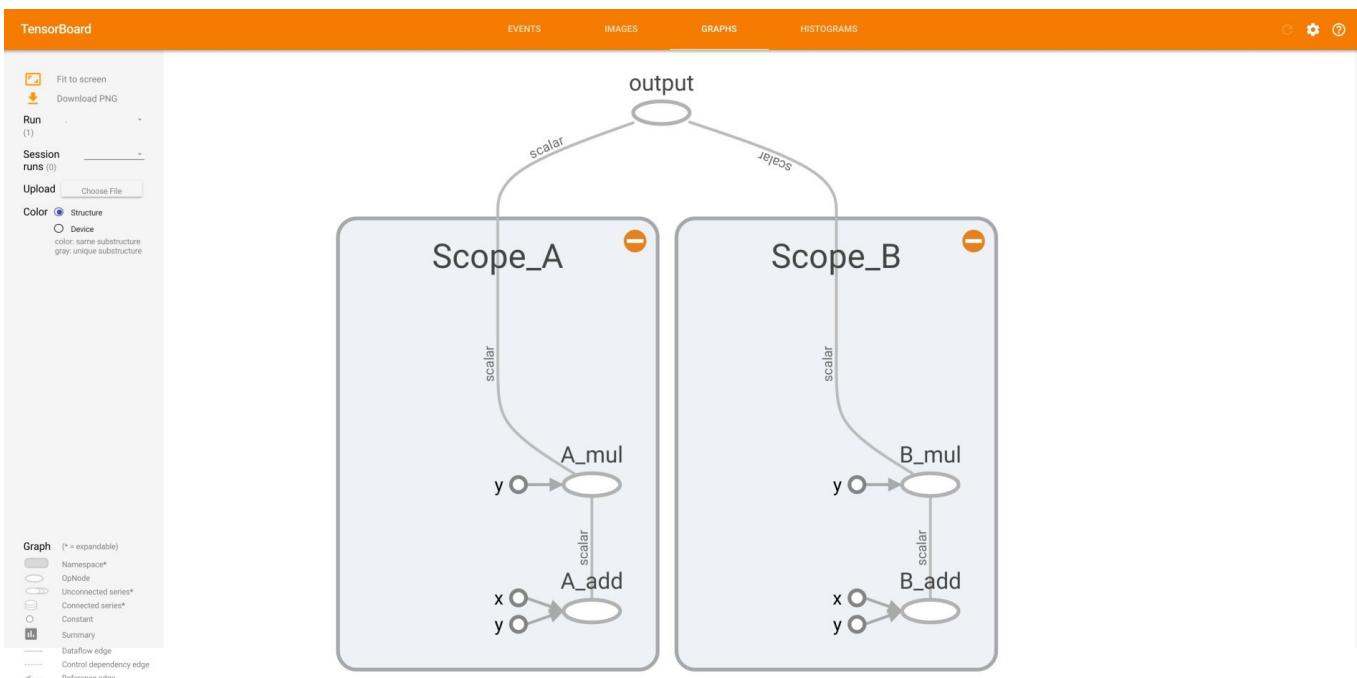
Because the `summarywriter` exports the graph immediately, we can simply start up TensorBoard after running the above code. Navigate to where you ran the previous script and start up TensorBoard:

```
$ tensorboard --logdir='./name_scope_1'
```

As before, this will start a TensorBoard server on your local computer at port 6006. Open up a browser and enter `localhost:6006` into the URL bar. Navigate to the "Graph" tab, and you'll see something similar to this:



You'll notice that the `add` and `mul` Operations we added to the graph aren't immediately visible- instead, we see their enclosing name scopes. You can expand the name scope boxes by clicking on the plus + icon in their upper right corner.



Inside of each scope, you'll see the individual Operations you've added to the graph. You can also nest name scopes within other name scopes:

```
graph = tf.Graph()

with graph.as_default():
    in_1 = tf.placeholder(tf.float32, shape=[], name="input_a")
    in_2 = tf.placeholder(tf.float32, shape=[], name="input_b")
    const = tf.constant(3, dtype=tf.float32, name="static_value")

    with tf.name_scope("Transformation"):

        with tf.name_scope("A"):
            A_mul = tf.mul(in_1, const)
            A_out = tf.sub(A_mul, in_1)

        with tf.name_scope("B"):
```

```

B_mul = tf.mul(in_2, const)
A_out = tf.sub(B_mul, in_2)

with tf.name_scope("C"):
    C_div = tf.div(A_out, B_out)
    C_out = tf.add(C_div, const)

with tf.name_scope("D"):
    D_div = tf.div(B_out, A_out)
    D_out = tf.add(D_div, const)

out = tf.maximum(C_out, D_out)

writer = tf.train.SummaryWriter('./name_scope_2', graph=graph)
writer.close()

```

To mix things up, this code explicitly creates a `tf.Graph` object instead of using the default graph. Let's look at the code and focus on the name scopes to see exactly how it's structured:

```

graph = tf.Graph()

with graph.as_default():
    in_1 = tf.placeholder(...)
    in_2 = tf.placeholder(...)
    const = tf.constant(...)

    with tf.name_scope("Transformation"):

        with tf.name_scope("A"):
            # Takes in_1, outputs some value
            ...

        with tf.name_scope("B"):
            # Takes in_2, outputs some value
            ...

        with tf.name_scope("C"):
            # Takes the output of A and B, outputs some value
            ...

        with tf.name_scope("D"):
            # Takes the output of A and B, outputs some value
            ...

    # Takes the output of C and D
    out = tf.maximum(...)

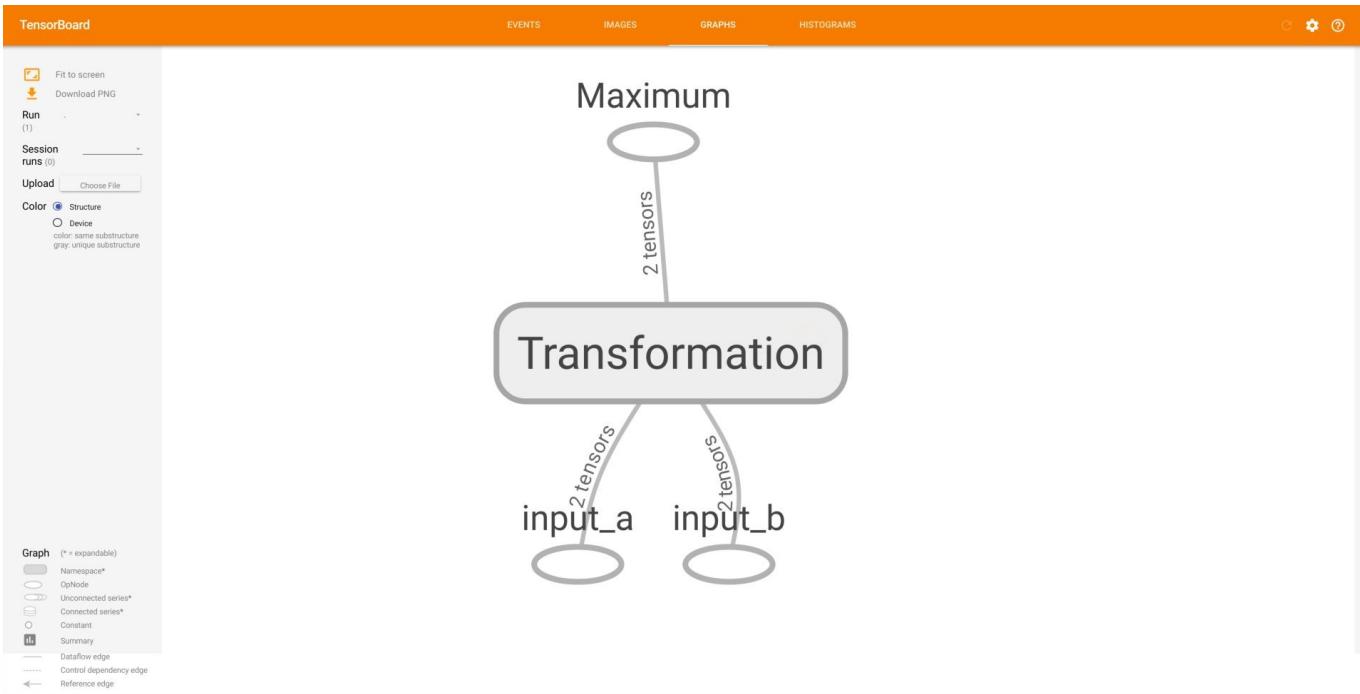
```

Now it's easier to dissect. This model has two scalar placeholder nodes as input, a TensorFlow constant, a middle chunk called "Transformation", and then a final output node that uses `tf.maximum()` as its Operation. We can see this high-level overview inside of TensorBoard:

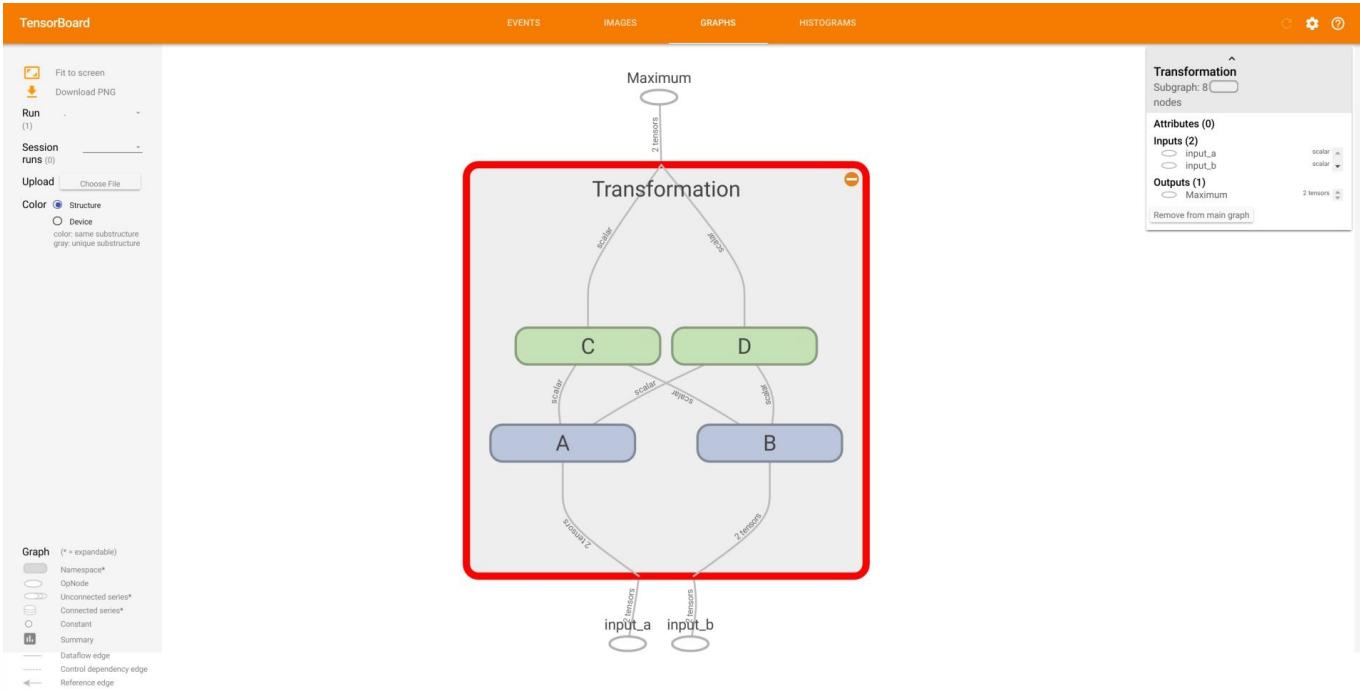
```

# Start up TensorBoard in a terminal, loading in our previous graph
$ tensorboard --logdir='./name_scope_2'

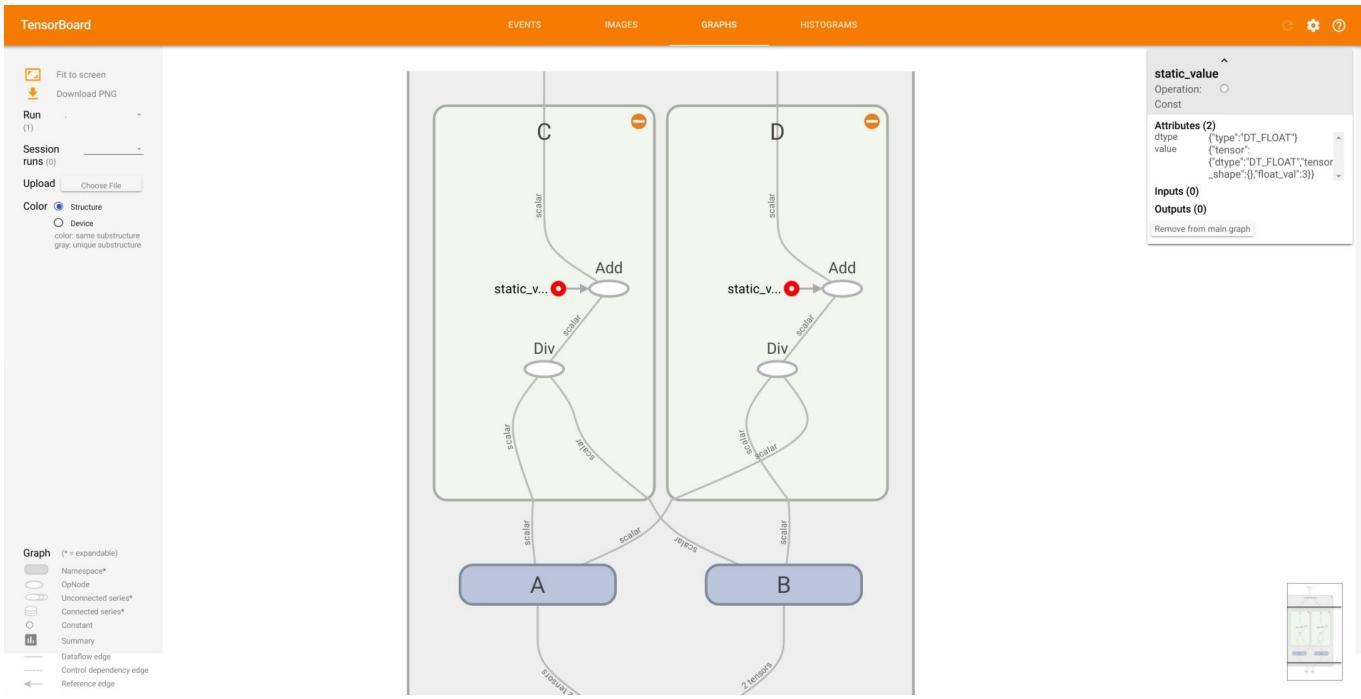
```



Inside of the Transformation name scope are four more name scopes arranged in two “layers”. The first layer is comprised of scopes “A” and “B”, which pass their output values into the next layer of “C” and “D”. The final node then uses the outputs from this last layer as its input. If you expand the Transformation name scope in TensorBoard, you’ll get a look at this:



This also gives us a chance to showcase another feature in TensorBoard. In the above picture, you'll notice that name scopes "A" and "B" have matching color (blue), as do "C" and "D" (green). This is due to the fact that these name scopes have identical Operations setup in the same configuration. That is, "A" and "B" both have a `tf.mul()` Op feeding into a `tf.sub()` Op, while "C" and "D" have a `tf.div()` Op that feeds into a `tf.add()` Op. This becomes handy if you start using functions to create repeated sequences of Operations.



In this image, you can see that `tf.constant` objects don't behave quite the same way as other Tensors or Operations when displayed in TensorBoard. Even though we declared `static_value` outside of any name scope, it still gets placed inside them. Furthermore, instead of there being one icon for `static_value`, it creates a small visual whenever it is used. The basic idea for this is that constants can be used at any time and don't necessarily need to be used in any particular order. To prevent arrows flying all over the graph from a single point, it just makes a little small impression whenever a constant is used.

ASIDE: `tf.maximum()`

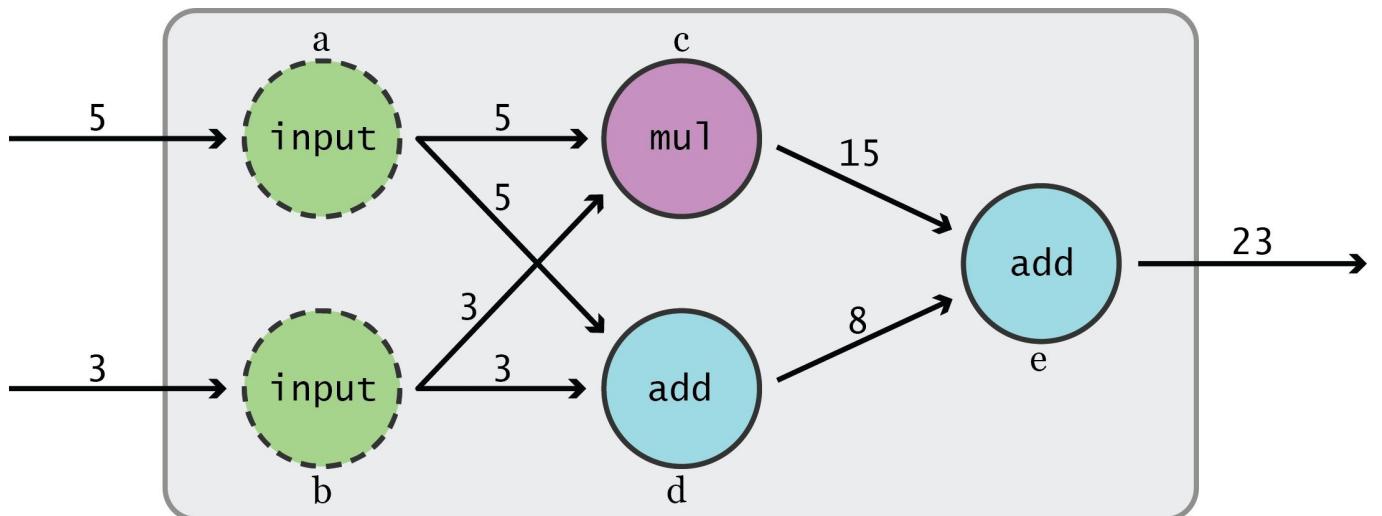
Separating a huge graph into meaningful clusters can make understanding and debugging your model a much more approachable task.

Logging with TensorBoard

Exercise: Putting it together

We'll end this chapter with an exercise that uses all of the components we've discussed: Tensors, Graphs, Operations, Variables, placeholders, Sessions, and name scopes. We'll also include some TensorBoard summaries so we can keep track of the graph as it runs. By the end of this, you should feel comfortable composing basic TensorFlow graphs and exploring it in TensorBoard.

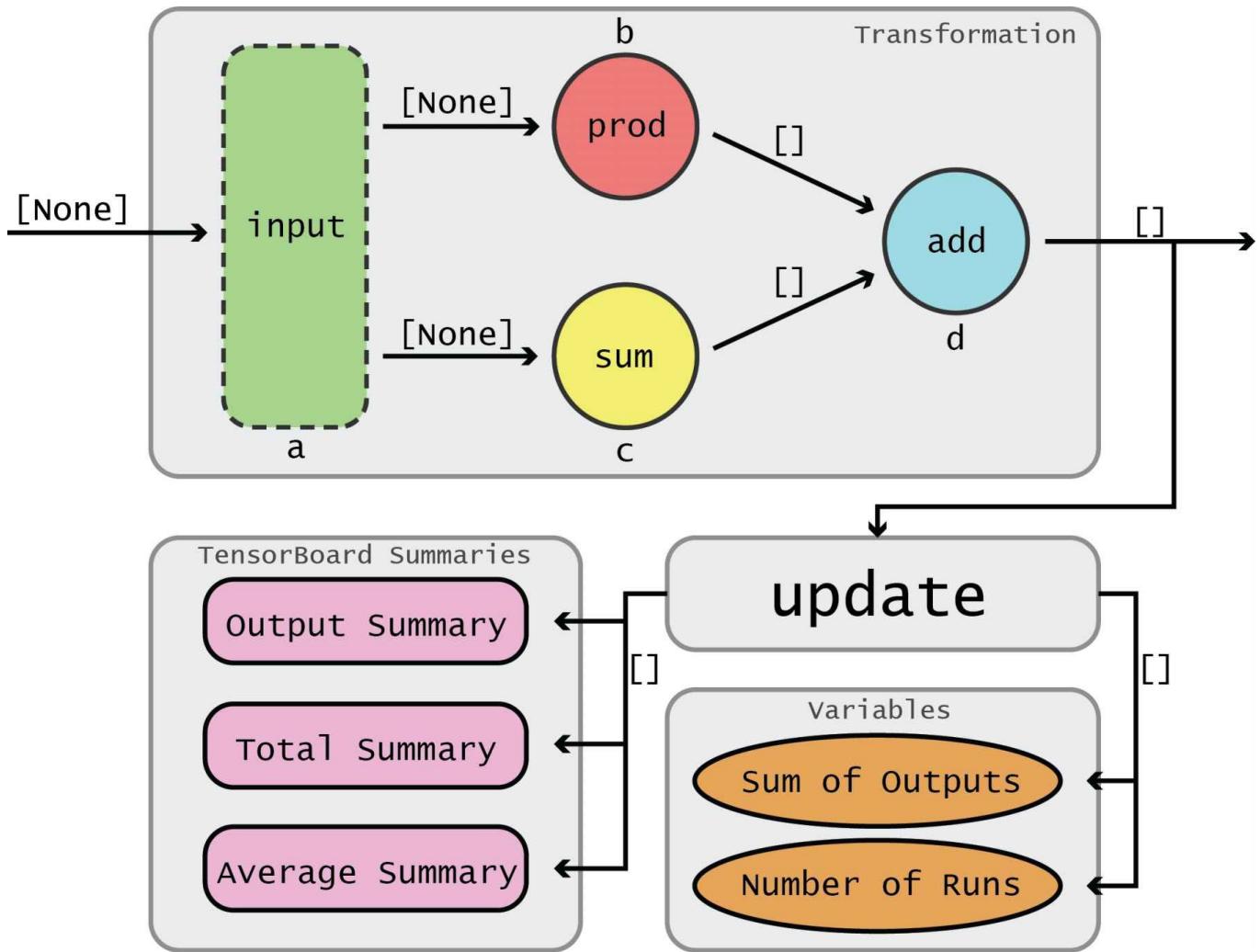
At its core, it is going to be the same sort of transformation as our first basic model:



But, this time it's going to have some important differences that use TensorFlow more fully:

- Our inputs will be placeholders instead of `tf.constant` nodes
- Instead of taking two discrete scalar inputs, our model will take in a single vector of any length
- We're going to accumulate the total value of all outputs over time as we use the graph
- The graph will be segmented nicely with name scopes
- After each run, we are going to save the output of the graph, the accumulated total of all outputs, and the average value of all outputs to disk for use in TensorBoard

Here's a rough outline of what we'd like our graph to look like:



Here are the key things to note about reading this model:

- Notice how each edge now has either a `[None]` or `[]` icon next to it. This represents the TensorFlow shape of the tensor flowing through that edge, with `None` representing a vector of any length, and `[]` representing a scalar.
- The output of node `d` now flows into an “update” section, which contains Operations necessary to update Variables as well as pass data through to the TensorBoard summaries.
- We have a separate name scope to contain our two Variables- one to store the accumulated sum of our outputs, the other to keep track of how many times we’ve run the graph. Since these two Variables operate outside of the flow of our main transformation, it makes sense to put them in a separate space.
- There is a name scope dedicated to our TensorBoard summaries which will hold our `tf.scalar_summary` Operations. We place them after the “update” section to ensure that the summaries are added *after* we update our Variables, otherwise things could run out of order.

Let’s get going! Open up your code editor or interactive Python environment.

Building the graph

The first thing we'll need to do, as always, is import the TensorFlow library:

```
import tensorflow as tf
```

We're going to explicitly create the graph that we'd like to use instead of using the default graph, so make one with `tf.Graph()`:

```
graph = tf.Graph()
```

And then we'll set our new graph as the default while we construct our model:

```
with graph.as_default():
```

We have two “global” style Variables in our model. The first is a “global_step”, which will keep track of how many times we've run our model. This is a common paradigm in TensorFlow, and you'll see it used throughout the API. The second Variable is called “total_output”- it's going to keep track of the total sum of all outputs run on this model over time. Because these Variables are global in nature, we declare them separately from the rest of the nodes in the graph and place them into their own name scope:

```
with graph.as_default():
```

```
    with tf.name_scope("variables"):
        # Variable to keep track of how many times the graph has been run
        global_step = tf.Variable(0, dtype=tf.int32, trainable=False, name="global_step")

        # Variable that keeps track of the sum of all output values over time:
        total_output = tf.Variable(0.0, dtype=tf.float32, trainable=False, name="total_output")
```

Note that we use the `trainable=False` setting- it won't have an impact in this model (we aren't training anything!), but it makes it explicit that these Variables will be set by hand.

Next up, we'll create the core transformation part of the model. We'll encapsulate the entire transformation in a name scope, “transformation”, and separate them further into separate “input”, “intermediate_layer”, and “output” name scopes:

```
with graph.as_default():
    with tf.name_scope("variables"):
        ...

    with tf.name_scope("transformation"):

        # Separate input layer
        with tf.name_scope("input"):
            # Create input placeholder- takes in a Vector
            a = tf.placeholder(tf.float32, shape=[None], name="input_placeholder_a")

        # Separate middle layer
        with tf.name_scope("intermediate_layer"):
            b = tf.reduce_prod(a, name="product_b")
            c = tf.reduce_sum(a, name="sum_c")

        # Separate output layer
        with tf.name_scope("output"):
            output = tf.add(b, c, name="output")
```

This is extremely similar to the code written for the previous model, with a few key differences:

- Our input node is a `tf.placeholder` that accepts a vector of any length (`shape=[None]`).
- Instead of using `tf.mul()` and `tf.add()`, we use `tf.reduce_prod()` and `tf.reduce_sum()`,

respectively. This allows us to multiply and add across the entire input vector, as the earlier Ops only accept *exactly* 2 input scalars.

After the transformation computation, we're going to need to update our two Variables from above. Let's create an "update" name scope to hold these changes:

```
with graph.as_default():
    with tf.name_scope("variables"):
        ...
    with tf.name_scope("transformation"):
        ...

    with tf.name_scope("update"):
        # Increments the total_output Variable by the latest output
        update_total = total_output.assign_add(output)

        # Increments the above `global_step` Variable, should be run whenever the graph is run
        increment_step = global_step.assign_add(1)
```

We use the `Variable.assign_add()` Operation to increment both `total_output` and `global_step`. We add on the value of `output` to `total_output`, as we want to accumulate the sum of all outputs over time. For `global_step`, we simply increment it by one.

After we have updated our Variables, we can create the TensorBoard summaries we're interested in. We'll place those inside a name scope called "summaries":

```
with graph.as_default():
    ...
    with tf.name_scope("update"):
        ...

    with tf.name_scope("summaries"):
        avg = tf.div(update_total, tf.cast(increment_step, tf.float32), name="average")

        # Creates summaries for output node
        tf.scalar_summary(b'Output', output, name="output_summary")
        tf.scalar_summary(b'Sum of outputs over time', update_total, name="total_summary")
        tf.scalar_summary(b'Average of outputs over time', avg, name="average_summary")
```

The first thing we do inside of this section is compute the average output value over time. Luckily, we have the total value of all outputs with `total_output` (we use the output from `update_total` to make sure that the update happens before we compute `avg`), as well as the total number of times we've run the graph with `global_step` (same thing- we use the output of `increment_step` to make sure the graph runs in order). Once we have the average, we save `output`, `update_total` and `avg` with separate `tf.scalar_summary` objects.

To finish up the graph, we'll create our Variable initialization Operation as well as a helper node to group all of our summaries into one Op. Let's place these in a name scope called "global_ops":

```
with graph.as_default():
    ...
    with tf.name_scope("summaries"):
        ...

    with tf.name_scope("global_ops"):
        # Initialization Op
        init = tf.initialize_all_variables()
        # Merge all summaries into one Operation
        merged_summaries = tf.merge_all_summaries()
```

You may be wondering why we placed the `tf.merge_all_summaries()` Op here instead of the "summaries" name scope. While it doesn't make a huge difference here, placing

`merge_all_summaries()` with other global Operations is generally best practice. This graph only has one section for summaries, but you can imagine a graph having different summaries for Variables, Operations, name scopes, etc. By keeping `merge_all_summaries()` separate, it ensures that you'll be able to find the Operation without having to remember which specific "summary" code block you placed it in.

That's it for creating the graph! Now let's get things set up to execute the graph.

Running the graph

Let's open up a `Session` and launch the `Graph` we just made. We can also open up a `tf.train.SummaryWriter`, which we'll use to save our summaries. Here, we list `./improved_graph` as our destination folder for summaries to be saved.

```
sess = tf.Session(graph=graph)
writer = tf.train.SummaryWriter('./improved_graph', graph)
```

With a Session started, let's initialize our Variables before doing anything else:

```
sess.run(init)
```

To actually run our graph, let's create a helper function, `run_graph()` so that we don't have to keep typing the same thing over and over again. What we'd like is to pass in our input vector to the function, which will run the graph and save our summaries:

```
def run_graph(input_tensor):
    feed_dict = {a: input_tensor}
    _, step, summary = sess.run([output, increment_step, merged_summaries],
                                feed_dict=feed_dict)
    writer.add_summary(summary, global_step=step)
```

Here's the line-by-line breakdown of `run_graph()`:

1. First, we create a dictionary to use as a `feed_dict` in `Session.run()`. This corresponds to our `tf.placeholder` node, using its handle `a`.
2. Next, we tell our `Session` to run the graph using our `feed_dict`, and we want to make sure that we run `output`, `increment_step`, and our `merged_summaries` Ops. We need to save the `global_step` and `merged_summaries` values in order to write our summaries, so we save them to the `step` and `summary` Python variables. We use an underscore `_` to indicate that we don't care about storing the value of `output`.
3. Finally, we add the summaries to our `SummaryWriter`. The `global_step` parameter is important, as it allows TensorBoard to graph data over time (it essentially creates the x-axis on a line chart coming up).

Let's actually use it! Call `run_graph` several times with vectors of various lengths- like this:

```
run_graph([2,8])
run_graph([3,1,3,3])
run_graph([8])
run_graph([1,2,3])
run_graph([11,4])
run_graph([4,1])
run_graph([7,3,1])
run_graph([6,3])
run_graph([0,2])
run_graph([4,5,6])
```

Do it as many times as you'd like. Once you've had your fill, go ahead and write the summaries to disk with the `SummaryWriter.flush()` method:

```
writer.flush()
```

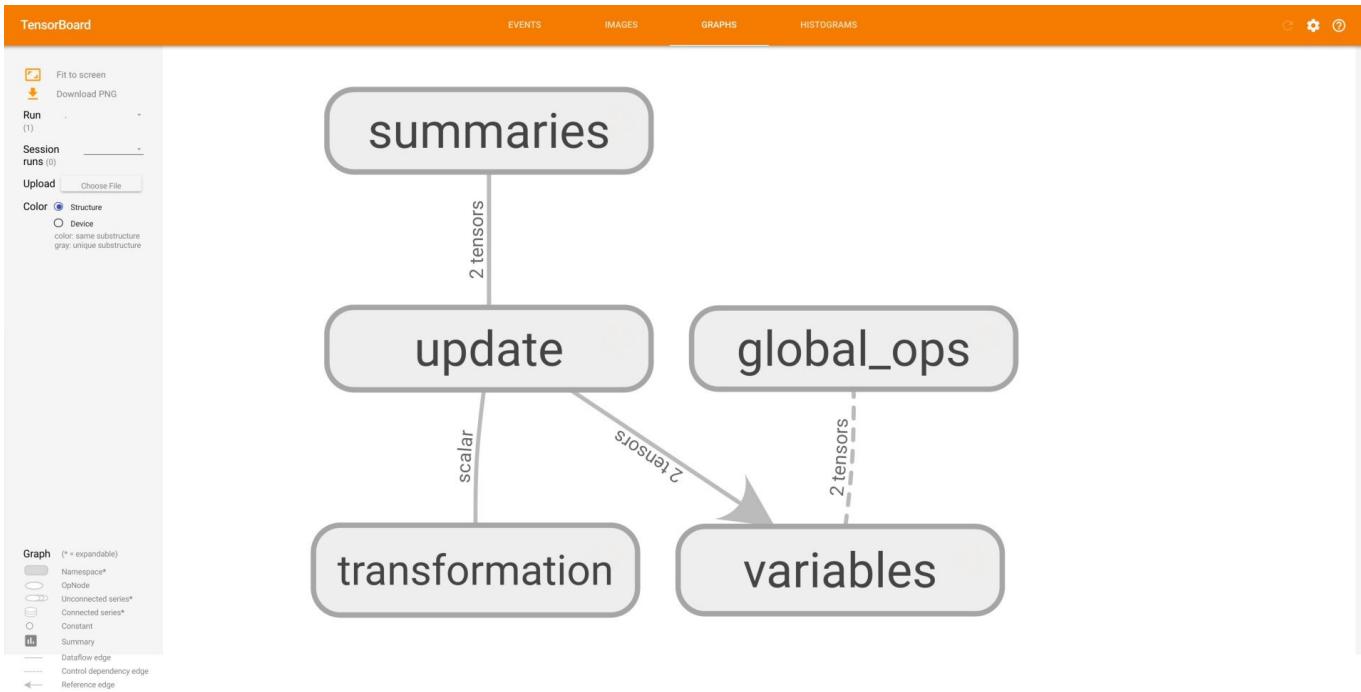
Finally, let's be tidy and close both our `SummaryWriter` and `Session`, now that we're done with them:

```
writer.close()
sess.close()
```

And that's it for our TensorFlow code! It was a little longer than the last graph, but it wasn't too bad. Let's open up TensorBoard and see what we've got. Fire up a terminal shell, navigate to the directory where you ran this code (make sure the "improved_graph" directory is located there), and run the following:

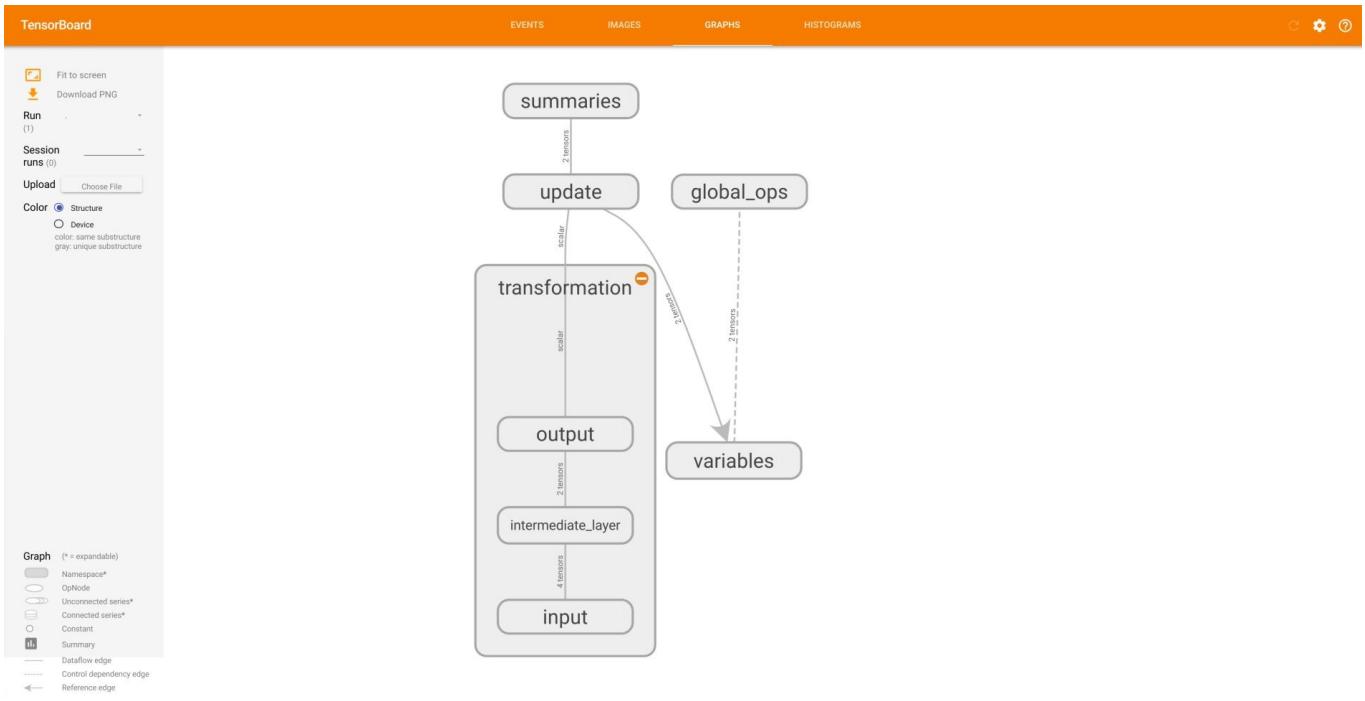
```
$ tensorboard --logdir="improved_graph"
```

As usual, this starts up a TensorBoard server on port 6006, hosting the data stored in "improved_graph". Type in "localhost:6006" into your web browser and let's see what we've got! Let's first check out the "Graph" tab:

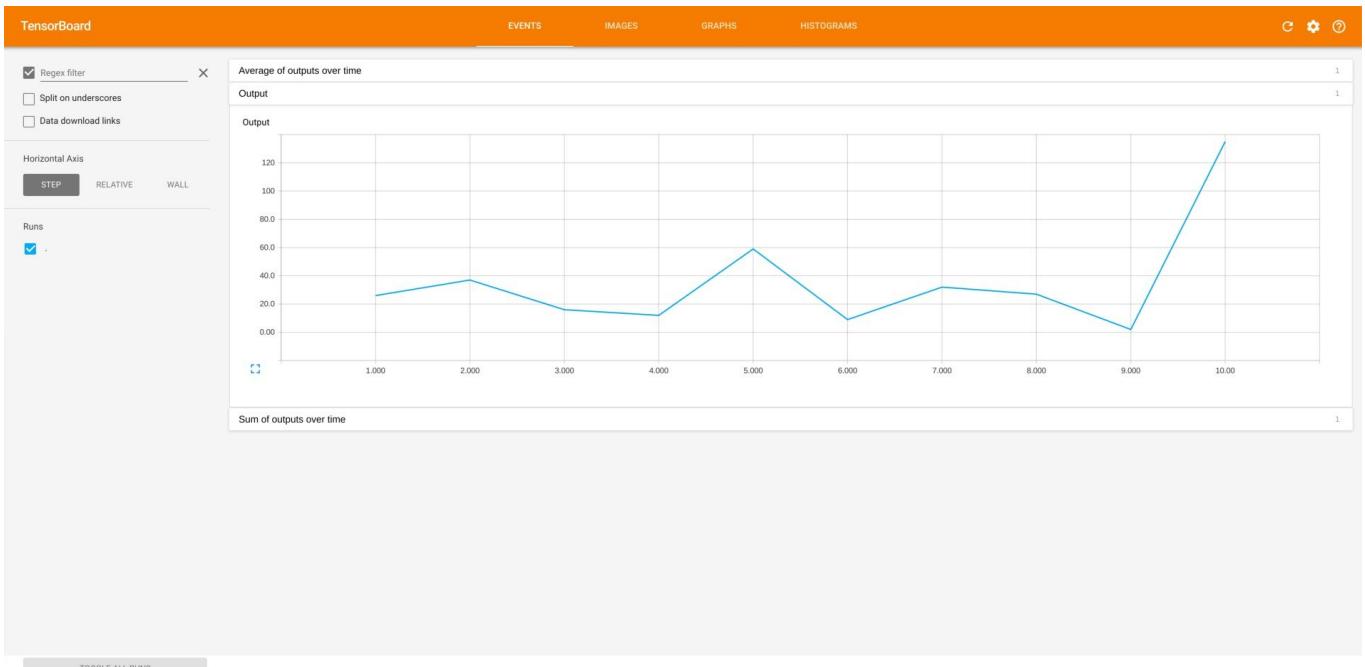


You'll see that this graph closely matches what we diagrammed out earlier. Our transformation operations flow into the update block, which then feeds into both the summary and variable name scopes. The main difference between this and our diagram is the "global_ops" name scope, which contains operations that aren't critical to the primary transformation computation.

You can expand the various blocks to get a more granular look at their structure:



Now we can see the separation of our input, the intermediate layer, and the output. It might be overkill on a simple model like this, but this sort of compartmentalization is extremely useful. Feel free to explore the rest of the graph. When you're ready, head over to the “Events” page.



When you open up the “Events” page, you should see three collapsed tabs, named based on the tags we gave our `scalar_summary` objects above. By clicking on any of them, you’ll see a nice line chart showing the values we stored at various time steps. If you click the blue rectangle at the bottom left of the charts, they’ll expand to look like the image above.

Definitely check out the results of your summaries, compare them, make sure that they make sense, and pat yourself on the back! That concludes this exercise- hopefully by now you have a good sense of how to create TensorFlow graphs based on visual sketches, as well as how to do some basic summaries with TensorBoard.

The entirety of the code for this exercise is below:

```
import tensorflow as tf

# Explicitly create a Graph object
graph = tf.Graph()

with graph.as_default():

    with tf.name_scope("variables"):
        # Variable to keep track of how many times the graph has been run
        global_step = tf.Variable(0, dtype=tf.int32, trainable=False, name="global_step")

        # Variable that keeps track of the sum of all output values over time:
        total_output = tf.Variable(0.0, dtype=tf.float32, trainable=False, name="total_output")

    # Primary transformation Operations
    with tf.name_scope("transformation"):

        # Separate input layer
        with tf.name_scope("input"):
            # Create input placeholder- takes in a Vector
            a = tf.placeholder(tf.float32, shape=[None], name="input_placeholder_a")

        # Separate middle layer
        with tf.name_scope("intermediate_layer"):
            b = tf.reduce_prod(a, name="product_b")
            c = tf.reduce_sum(a, name="sum_c")

        # Separate output layer
        with tf.name_scope("output"):
            output = tf.add(b, c, name="output")

    with tf.name_scope("update"):
        # Increments the total_output Variable by the latest output
        update_total = total_output.assign_add(output)

        # Increments the above `global_step` Variable, should be run whenever the graph is run
        increment_step = global_step.assign_add(1)

    # Summary Operations
    with tf.name_scope("summaries"):
        avg = tf.div(update_total, tf.cast(increment_step, tf.float32), name="average")

        # Creates summaries for output node
        tf.scalar_summary(b'Output', output, name="output_summary")
        tf.scalar_summary(b'Sum of outputs over time', update_total, name="total_summary")
        tf.scalar_summary(b'Average of outputs over time', avg, name="average_summary")

    # Global Variables and Operations
    with tf.name_scope("global_ops"):
        # Initialization Op
        init = tf.initialize_all_variables()
        # Merge all summaries into one Operation
        merged_summaries = tf.merge_all_summaries()

# Start a Session, using the explicitly created Graph
sess = tf.Session(graph=graph)

# Open a SummaryWriter to save summaries
writer = tf.train.SummaryWriter('./improved_graph', graph)

# Initialize Variables
sess.run(init)

def run_graph(input_tensor):
    """
    Helper function; runs the graph with given input tensor and saves summaries
    """
    feed_dict = {a: input_tensor}
    _, step, summary = sess.run([output, increment_step, merged_summaries],
                               feed_dict=feed_dict)
    writer.add_summary(summary, global_step=step)

# Run the graph with various inputs
```

```
run_graph([2,8])
run_graph([3,1,3,3])
run_graph([8])
run_graph([1,2,3])
run_graph([11,4])
run_graph([4,1])
run_graph([7,3,1])
run_graph([6,3])
run_graph([0,2])
run_graph([4,5,6])

# Write the summaries to disk
writer.flush()

# Close the SummaryWriter
writer.close()

# Close the session
sess.close()
```

Conclusion

That wraps up this chapter! There was a lot of information to absorb, and you should definitely play around with TensorFlow now that you have a grasp of the fundamentals. Get yourself fluent with Operations, Variables, and Sessions, and embed the basic loop of building and running graphs into your head.

Using TensorFlow for simple math problems is fun (for some people), but we haven't even touched on the primary use case for the library yet: machine learning. In the next chapter, you'll be introduced to some of the core concepts and techniques for machine learning and how to use them inside of TensorFlow.

Chapter 4. Machine Learning Basics

Now that we covered the basics about how Tensorflow works, we are ready to talk about its main usage: machine learning.

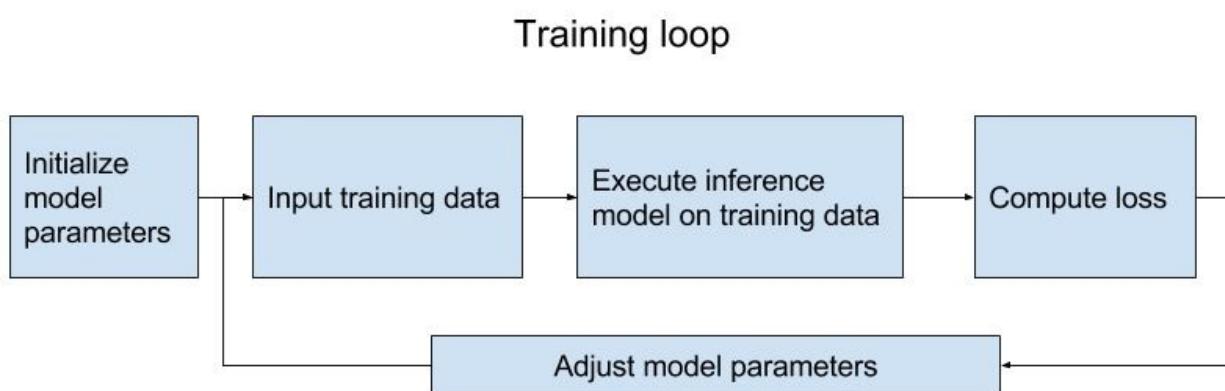
We are going to present high level notions of basic machine learning topics along with code snippets, showing how we work with them in Tensorflow.

Supervised learning introduction

In this book we will focus on supervised learning problems, where we **train an inference model** with an input dataset, along with the real or expected output for each example. The model will cover a dataset and then be able to predict the output for new inputs that don't exist in the original training dataset.

An inference model is a series of mathematical operations that we apply to our data. The steps are set by code and determined by the model we are using to solve a given problem. The operations composing our model are fixed. Inside the operations we have arbitrary values, like "multiply by 3" or "add 2." These values are the **parameters** of the model, and are the ones that change through training in order for the model to learn and adjust its output.

Although the inference models may vary significantly in the number of operations they use, and in the way they combine and the number of parameters they have; we always apply the same general structure for training them:



We create a **training loop** that:

- Initializes the model parameters for the first time. Usually we use random values for this, but in simple models we may just set zeros.
- Reads the training data along with the expected output data for each data example. Usual operations here may also imply randomizing the order of the data for always feeding it differently to the model.
- Executes the inference model on the training data, so it calculates for each training input example the output with the current model parameters.
- Computes the loss. The loss is a single value that will summarize and indicate to our model how far are the values that computed in the last step with the expected output from the training set. There are different loss functions that you can use and are present in the book.
- Adjusts the model parameters. This is where the learning actually takes place. Given the loss function, learning is just a matter of improving the values of the parameters

in order to minimize the loss through a number of training steps. Most commonly, you can use a gradient descent algorithm for this, which we will explain in the following section.

The loop repeats this process through a number of cycles, according to the learning rate that we need to apply, and depending on the model and data we input to it.

After training, we apply an *evaluation phase*; where we execute the inference against a different set data to which we also have the expected output, and evaluate the loss for it. Given how this dataset contains examples unknown for the model, the evaluation tells you how well the model predicts beyond its training. A very common practice is to take the original dataset and randomly split it in 70% of the examples for training, and 30% for evaluation.

Let's use this structure to define some generic scaffolding for the model code.

```
import tensorflow as tf

# initialize variables/model parameters

# define the training loop operations
def inference(X):
    # compute inference model over data X and return the result

def loss(X, Y):
    # compute loss over training data X and expected outputs Y

def inputs():
    # read/generate input training data X and expected outputs Y

def train(total_loss):
    # train / adjust model parameters according to computed total loss

def evaluate(sess, X, Y):
    # evaluate the resulting trained model

# Launch the graph in a session, setup boilerplate
with tf.Session() as sess:
    tf.initialize_all_variables().run()

    X, Y = inputs()

    total_loss = loss(X, Y)
    train_op = train(total_loss)

    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)

    # actual training loop
    training_steps = 1000
    for step in range(training_steps):
        sess.run([train_op])
        # for debugging and learning purposes, see how the loss gets decremented thru training steps
        if step % 10 == 0:
            print "loss: ", sess.run([total_loss])

    evaluate(sess, X, Y)

    coord.request_stop()
    coord.join(threads)
    sess.close()
```

This is the basic shape for an inference model code. First it initializes the model parameters. Then it defines a method for each of the training loop operations: read input training data (`inputs` method), compute inference model (`inference` method), calculate loss

over expected output (`loss` method), adjust model parameters (`train` method), evaluate the resulting model (`evaluate` method), and then the boilerplate code to start a session and run the training loop. In the following sections we will fill these template methods with required code for each type of inference model.

Once you are happy with how the model responds, you can focus on exporting it and serving it to run inference against the data you need to work with.

Saving training checkpoints

As we stated above, training models implies updating their parameters, or variables in Tensorflow lingo, through many training cycles. Variables are stored in memory, so if the computer would lose power after many hours of training, we would lose all of that work. Luckily, there is the `tf.train.Saver` class to save the graph variables in proprietary binary files. We should periodically save the variables, create a *checkpoint* file, and eventually restore the training from the most recent checkpoint if needed.

In order to use the `Saver` we need to slightly change the training loop scaffolding code:

```
# model definition code...

# Create a saver.
saver = tf.train.Saver()

# Launch the graph in a session, setup boilerplate
with tf.Session() as sess:

    # model setup....

    # actual training loop
    for step in range(training_steps):
        sess.run([train_op])

        if step % 1000 == 0:
            saver.save(sess, 'my-model', global_step=step)

    # evaluation...
    saver.save(sess, 'my-model', global_step=training_steps)

sess.close()
```

In the code above we instantiate a saver before opening the session, inserting code inside the training loop to call the `tf.train.Saver.save` method for each 1000 training steps, along with the final step when the training loop finishes. Each call will create a checkpoint file with the name template `my-model-{step}` like `my-model-1000`, `my-model-2000`, etc. The file stores the current values of each variable. By default the saver will keep only the most recent 5 files and delete the rest.

If we wish to recover the training from a certain point, we should use the `tf.train.get_checkpoint_state` method, which will verify if we already have a checkpoint saved, and the `tf.train.Saver.restore` method to recover the variable values.

```
with tf.Session() as sess:

    # model setup...

    initial_step = 0

    # verify if we don't have a checkpoint saved already
    ckpt = tf.train.get_checkpoint_state(os.path.dirname(__file__))
    if ckpt and ckpt.model_checkpoint_path:
        # Restores from checkpoint
        saver.restore(sess, ckpt.model_checkpoint_path)
        initial_step = int(ckpt.model_checkpoint_path.rsplit('-', 1)[1])

    # actual training loop
    for step in range(initial_step, training_steps):
        ...
```

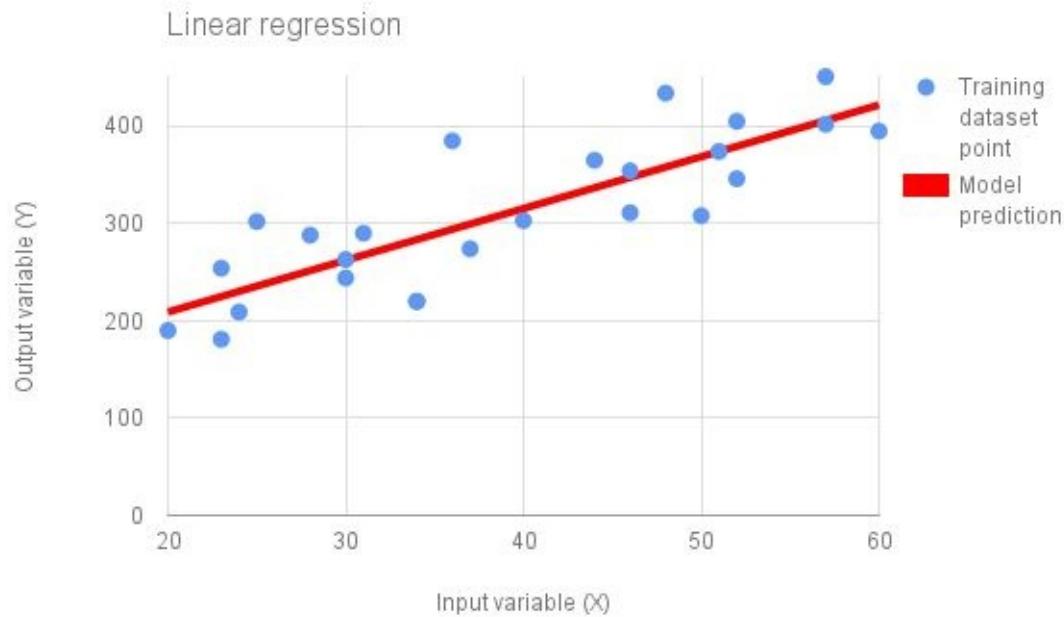
In the code above we check first if we have a checkpoint file, and then restore the

variable values before starting the training loop. We also recover the global step number from the checkpoint file name.

Now that we know how supervised learning works in general, as well as how to store our training progress, let's move on to explain some inference models.

Linear regression

Linear regression is the simplest form of modeling for a supervised learning problem. Given a set of data points as training data, you are going to find the linear function that best fits them. In a 2-dimensional dataset, this type of function represents a straight line.



Here is the charting of the lineal regression model in 2D. Blue dots are the training data points and the red line is the what the model will infer.

Let's begin with a bit of math to explain how the model will work. The general formula of a linear function is:

$$y(x_1, x_2, \dots, x_k) = w_1x_1 + w_2x_2 + \dots + w_kx_k + b$$

And its matrix (or tensor) form:

$$Y = XW^T + b \text{ where } X = (x_1, \dots, x_k) \text{ and } W = (w_1, \dots, w_k)$$

- Y is the value we are trying to predict.
- x_1, \dots, x_k independent or predictor variables are the values that we provide when using our model for predicting new values. In matrix form, you can provide multiple examples at once- one per row.
- w_1, \dots, w_k are the parameters the model will learn from the training data, or the “weights” given to each variable.
- b is also a learned parameter- the constant of the linear function that is also known as the bias of the model.

Let's represent this model in code. Instead of transposing weights, we can simply define them as a single column vector.

```
# initialize variables/model parameters
```

```
W = tf.Variable(tf.zeros([2, 1]), name="weights")
b = tf.Variable(0., name="bias")

def inference(X):
    return tf.matmul(X, W) + b
```

Now we have to define how to compute the loss. For this simple model we will use a squared error, which sums the squared difference of all the predicted values for each training example with their corresponding expected values. Algebraically it is the squared euclidean distance between the predicted output vector and the expected one. Graphically in a 2d dataset is the length of the vertical line that you can trace from the expected data point to the predicted regression line. It is also known as L2 norm or L2 loss function. We use it squared to avoid computing the square root, since it makes no difference for trying to minimize the loss and saves us a computing step.

$$loss = \sum_i (y_i - y_{predicted_i})^2$$

We sum over i , where i is each data example. In code:

```
def loss(X, Y):  
    Y_predicted = inference(X)  
    return tf.reduce_sum(tf.squared_difference(Y, Y_predicted))
```

It's now time to actually train our model with data. As an example, we are going to work with a dataset that relates age in years and weight in kilograms with blood fat content (<http://people.sc.fsu.edu/~jb Burkardt/datasets/regression/x09.txt>).

As the dataset is short enough for our example, we are just going to embed it in our code. In the following section we will show how to deal with reading the training data from files, like you would in a real world scenario.

```
def inputs():
    weight_age = [[84, 46], [73, 20], [65, 52], [70, 30], [76, 57], [69, 25], [63, 28], [72, 36], [7
    blood_fat_content = [354, 190, 405, 263, 451, 302, 288, 385, 402, 365, 209, 290, 346, 254, 395,
    return tf.to_float(weight_age), tf.to_float(blood_fat_content)
```

And now we define the model training operation. We will use the *gradient descent* algorithm for optimizing the model parameters, which we describe in the following section.

```
def train(total_loss):
    learning_rate = 0.0000001
    return tf.train.GradientDescentOptimizer(learning_rate).minimize(total_loss)
```

When you run it, you will see printed how the loss gets smaller on each training step.

Now that we trained the model, it's time to evaluate it. Let's compute the expected blood fat for a 25 year old person who weighs 80 kilograms. This is not originally in the source data, but we will compare it with another person with the same age who weighs 65 kilograms:

```
def evaluate(sess, X, Y):
    print sess.run(inference([[80., 25.]])) # ~ 303
    print sess.run(inference([[65., 25.]])) # ~ 256
```

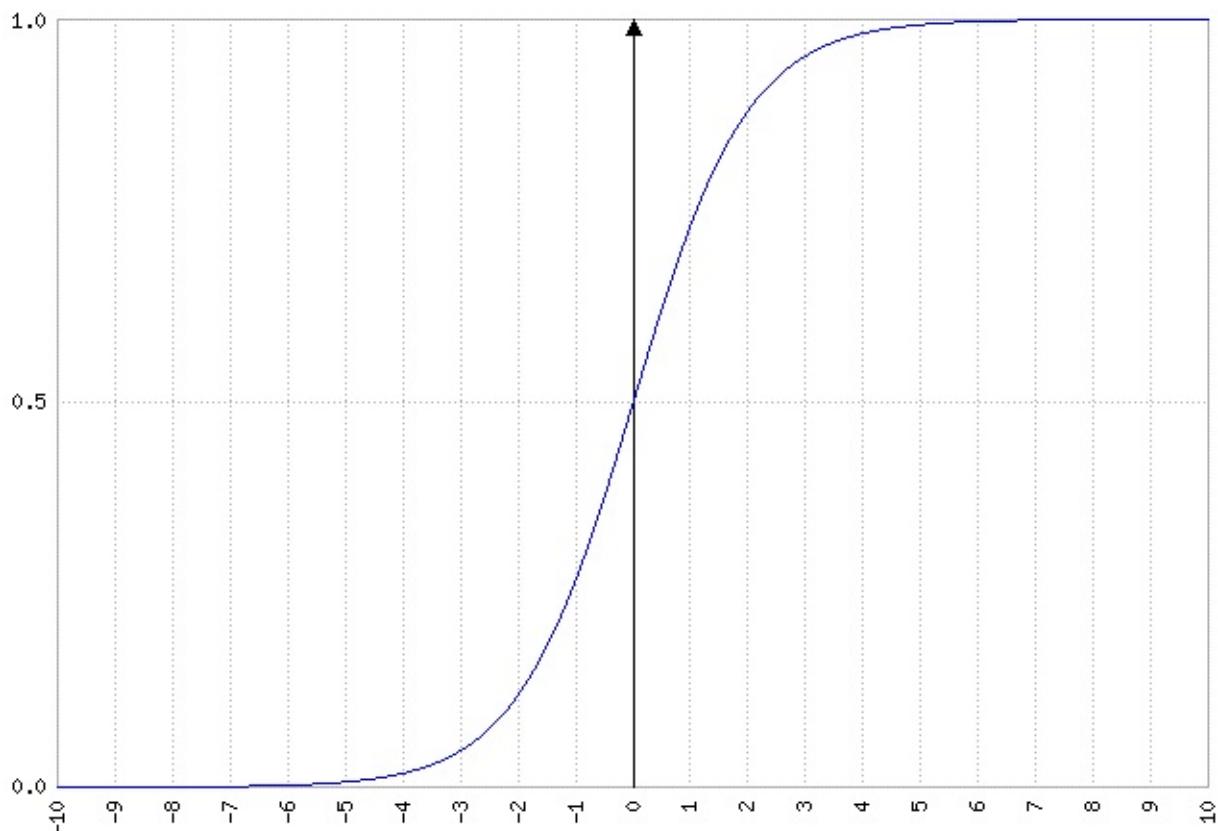
As a quick evaluation, you can check that the model learned how the blood fat decays with weight, and the output values are in between the boundaries of the original trained values.

Logistic regression

The linear regression model predicts a *continuous* value, or any real number. Now we are going to present a model that can answer a yes-no type of question, like “Is this email spam?”

There is a function used commonly in machine learning called the **logistic function**. It is also known as the *sigmoid* function, because its shape is an S (and sigma is the greek letter equivalent to s).

$$f(x) = \frac{1}{1 + e^{-x}}$$



Here you see the charting of a logistic/sigmoid function, with its “S” shape.

The logistic function is a probability distribution function that, given a specific input value, computes the probability of the output being a *success*, and thus the probability for the answer to the question to be “yes.”

This function takes a single input value. In order to feed the function with the multiple dimensions, or features from the examples of our training datasets, we need to combine them into a single value. We can use the linear regression model expression for doing this, like we did in the section above.

To express it in code, you can reuse all of the elements of the linear model, however, you just slightly change the prediction to apply the sigmoid.

```

# same params and variables initialization as log reg.
W = tf.Variable(tf.zeros([5, 1]), name="weights")
b = tf.Variable(0., name="bias")

# former inference is now used for combining inputs
def combine_inputs(X):
    return tf.matmul(X, W) + b

# new inferred value is the sigmoid applied to the former
def inference(X):
    return tf.sigmoid(combine_inputs(X))

```

Now let's focus on the loss function for this model. We could use the squared error. The logistic function computes the probability of the answer being "yes." In the training set, a "yes" answer should represent 100% of probability, or simply the output value to be 1. Then the loss should be how much probability our model assigned less than 1 for that particular example, squared. Consequently, a "no" answer will represent 0 probability, hence the loss is any probability the model assigned for that example, and again squared.

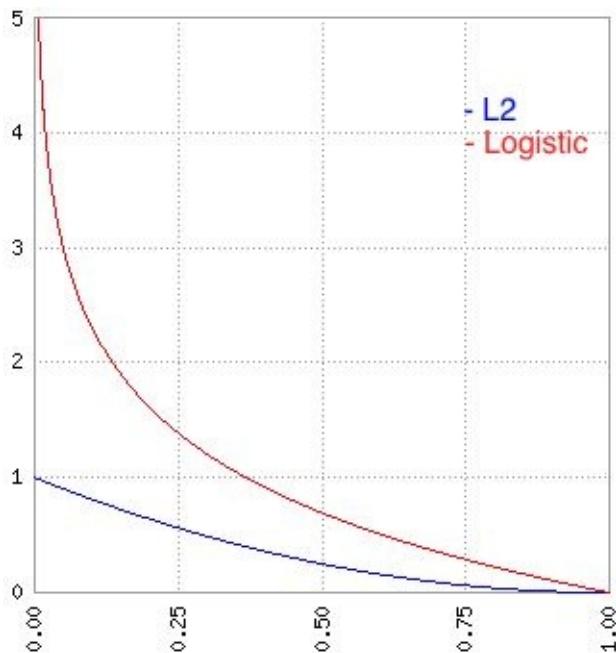
Consider an example where the expected answer is "yes" and the model is predicting a very low probability for it, close to 0. This means that it is close to 100% sure that the answer is "no."

The squared error penalizes such a case with the same order of magnitude for the loss as if the probability would have been predicted as 20, 30, or even 50% for the "no" output.

There is a loss function that works better for this type of problem, which is the **cross entropy** function.

$$loss = - \sum_i (y_i \cdot \log(y_{predicted_i}) + (1 - y_i) \cdot \log(1 - y_{predicted_i}))$$

We can visually compare the behavior of both loss functions according to the predicted output for a "yes."



The cross entropy and squared error (L2) functions are charted together. Cross entropy outputs a much greater value ("penalizes"), because the output is farther from what is

expected.

With cross entropy, as the predicted probability comes closer to 0 for the “yes” example, the penalty increases closer to infinity. This makes it impossible for the model to make that misprediction after training. That makes the cross entropy better suited as a loss function for this model.

There is a Tensorflow method for calculating cross entropy directly for a sigmoid output in a single, optimized step:

```
def loss(X, Y):  
    return tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(combine_inputs(X), Y))
```

What “cross-entropy” means

In information theory, Shannon entropy allows to estimate the average minimum number of bits needed to encode a symbol s_i from a string of symbols, based on the probability p_i of each symbol to appear in that string.

$$H = - \sum_i (p_i \cdot \log_2(p_i))$$

You can actually link this entropy with the thermodynamics concept of entropy, in addition to their math expressions being analogous.

For instance, let's calculate the entropy for the word “HELLO.”

$$p("H") = p("E") = p("O") = 1/5 = 0.2$$

$$p("L") = 2/5 = 0.4$$

$$H = - 3 * 0.2 * \log_2(0.2) - 0.4 * \log_2(0.4) = 1.92193$$

So you need 2 bits per symbol to encode “HELLO” in the optimal encoding.

If you encode the symbols assuming any other probability for the q_i than the real p_i need more bits for encoding each symbol. That's where cross-entropy comes to play. It allows you to calculate the average minimum number of bits needed to encode the same string in another suboptimal encoding.

$$H = - \sum_i (p_i \cdot \log_2(q_i))$$

For example, ASCII assigns the uniform probability $q_i = 1/256$ for all its symbols. Let's calculate the cross-entropy for the word “HELLO” in ASCII encoding.

$$q("H") = q("E") = q("L") = q("O") = 1/256$$

$$H = -3 * 0.2 * \log_2(1/256) - 0.4 * \log_2(1/256) = 8$$

So, you need 8 bits per symbol to encode “HELLO” in ASCII, as you would have expected.

As a loss function, consider P to be expected training output and distribution probability (“encoding”), where the actual value has 100% and any other 0. And use Q as the model calculated output. Remember that the sigmoid function computes a probability.

It is a theorem that cross entropy is at its minimum when $P = Q$. Thus, you can use cross entropy to compare how a distribution “fits” another. The closer the cross entropy is to the entropy, the better Q is an approximation of P . Then effectively, cross-entropy reduces as the model better resembles the expected output, like you need in a loss function.

We can freely exchange \log_2 with \log for minimizing the entropy as you switch one to another by multiplying by the change of the base constant.

Now let’s apply the model to some data. We are going to use the Titanic survivor Kaggle contest dataset from <https://www.kaggle.com/c/titanic/data>.

The model will have to infer, based on the passenger age, sex and ticket class if the passenger survived or not.

To make it a bit more interesting, let’s use data from a file this time. Go ahead and download the `train.csv` file.

Here are the code basics for reading the file. This is a new method for our scaffolding. You can load and parse it, and create a batch to read many rows packed in a single tensor for computing the inference efficiently.

```
def read_csv(batch_size, file_name, record_defaults):
    filename_queue = tf.train.string_input_producer([os.path.dirname(__file__) + "/" + file_name])

    reader = tf.TextLineReader(skip_header_lines=1)
    key, value = reader.read(filename_queue)

    # decode_csv will convert a Tensor from type string (the text line) in
    # a tuple of tensor columns with the specified defaults, which also
    # sets the data type for each column
    decoded = tf.decode_csv(value, record_defaults=record_defaults)

    # batch actually reads the file and loads "batch_size" rows in a single tensor
    return tf.train.shuffle_batch(decoded,
                                  batch_size=batch_size,
                                  capacity=batch_size * 50,
                                  min_after_dequeue=batch_size)
```

You have to use **categorical data** from this dataset. Ticket class and gender are string features with a predefined possible set of values that they can take. To use them in the inference model we need to convert them to numbers. A naive approach might be assigning a number for each possible value. For instance, you can use “1” for first ticket class, “2” for second, and “3” for third. Yet that forces the values to have a lineal

relationship between them that doesn't really exist. You can't say that "third class is 3 times first class". Instead you should expand each categorical feature to N boolean features, or one for each possible value of the original. This allows the model to learn the importance of each possible value independently. In our example data, "first class" should have greater probability of survival than others.

When working with categorical data, convert it to multiple boolean features, one for each possible value. This allows the model to weight each possible value separately.

In the case of categories with only two possible values, like the gender in the dataset, it is enough to have a single variable for it. That's because you can express a linear relationship between the values. For instance if possible values are `female = 1` and `male = 0`, then `male = 1 - female`, a single weight can learn to represent both possible states.

```
def inputs():
    passenger_id, survived, pclass, name, sex, age, sibsp, parch, ticket, fare, cabin, embarked =
        read_csv(100, "train.csv", [[0.0], [0.0], [0], [""], [""], [0.0], [0.0], [0.0], [""], [0.0],
    # convert categorical data
    is_first_class = tf.to_float(tf.equal(pclass, [1]))
    is_second_class = tf.to_float(tf.equal(pclass, [2]))
    is_third_class = tf.to_float(tf.equal(pclass, [3]))
    gender = tf.to_float(tf.equal(sex, ["female"]))

    # Finally we pack all the features in a single matrix;
    # We then transpose to have a matrix with one example per row and one feature per column.
    features = tf.transpose(tf.pack([is_first_class, is_second_class, is_third_class, gender, age]))
    survived = tf.reshape(survived, [100, 1])

    return features, survived
```

In the code above we define our inputs as calling `read_csv` and converting the data. To convert to boolean, we use the `tf.equal` method to compare equality to a certain constant value. We also have to convert the boolean back to a number to apply inference with `tf.to_float`. We then pack all the booleans in a single tensor with `tf.pack`.

Finally, lets train our model.

```
def train(total_loss):
    learning_rate = 0.01
    return tf.train.GradientDescentOptimizer(learning_rate).minimize(total_loss)
```

To evaluate the results we are going to run the inference against a batch of the training set and count the number of examples that were correctly predicted. We call that measuring the *accuracy*.

```
def evaluate(sess, X, Y):  
    predicted = tf.cast(inference(X) > 0.5, tf.float32)  
    print sess.run(tf.reduce_mean(tf.cast(tf.equal(predicted, Y), tf.float32)))
```

As the model computes a probability of the answer being yes, we convert that to a positive answer if the output for an example is greater than 0.5. Then we compare equality with the actual value using `tf.equal`. Finally, we use `tf.reduce_mean`, which counts all of the correct answers (as each of them adds 1) and divides by the total number of samples in the batch, which calculates the percentage of right answers.

If you run the code above you should get around 80% of accuracy, which is a good number for the simplicity of this model.

Softmax classification

With logistic regression we were able to model the answer to the yes-no question. Now we want to be able to answer a multiple choice type of question like: “Were you born in Boston, London, or Sydney?”

For that case there is the **softmax** function, which is a generalization of the logistic regression for C possible different values.

$$f(x)_c = \frac{e^{-x_c}}{\sum_{j=0}^{C-1} e^{-x_j}} \text{ for } c = 0 \dots C-1$$

It returns a probability vector of C components, filling the corresponding probability for each output class. As it is a probability, the sum of the C vector components always equal to 1. That is because the formula is composed such that every possible input data example must belong to one output class, covering the 100% of possible examples. If the sum would be less than 1, it would imply that there could be some hidden class alternative. If it would be more than 1, it would mean that each example could belong to more than one class.

You can proof that if the number of classes is 2, the resulting output probability is the same as a logistic regression model.

Now, to code this model, you will have one slight change from the previous models in the variable initialization. Given that our model computes C outputs instead of just one, we need to have C different weight groups, one for each possible output. So, you will use a weights matrix, instead of a weights vector. That matrix will have one row for each input feature, and one column for each output class.

We are going to use the classical Iris flower dataset for trying softmax. You can download it from <https://archive.ics.uci.edu/ml/datasets/Iris> It contains 4 data features and 3 possible output classes, one for each type of iris plant, so our weights matrix should have a 4x3 dimension.

The variable initialization code should look like:

```
# this time weights form a matrix, not a vector, with one "feature weights column" per output class.
W = tf.Variable(tf.zeros([4, 3]), name="weights")
# so do the biases, one per output class.
b = tf.Variable(tf.zeros([3]), name="bias"))
```

Also, as you would expect, Tensorflow contains an embedded implementation of the softmax function.

```
def inference(X):
    return tf.nn.softmax(combine_inputs(X))
```

Regarding loss computation, the same considerations for logistic regression apply for

fitting a candidate loss function, as the output here is also a probability. We are going to use then cross-entropy again, adapted for multiple classes in the computed probability.

For a single training example i , cross entropy now becomes:

$$loss_i = - \sum_c y_c \cdot \log(y_{predicted_c})$$

Summing the loss for each output class on that training example. Note that y_c would equal 1 for the expected class of the training example and 0 for the rest, so only one loss value is actually summed, the one measuring how far the model predicted the probability for the true class.

Now to calculate the total loss among the training set, we sum the loss for each training example:

$$loss = - \sum_i \sum_c y_{c_i} \cdot \log(y_{predicted_{c_i}})$$

In code, there are two versions implemented in Tensorflow for the softmax cross-entropy function: one specially optimized for training sets with a single class value per example. For example, our training data may have a class value that could be either “dog”, “person” or “tree”. That function is `tf.nn.sparse_softmax_cross_entropy_with_logits`.

```
def loss(X, Y):
    return tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(combine_inputs(X), Y))
```

The other version of it lets you work with training sets containing the probabilities of each example to belong to every class. For instance, you could use training data like “60% of the asked people consider that this picture is about dogs, 25% about trees, and the rest about a person”. That function is `tf.nn.softmax_cross_entropy_with_logits`. You may need such a function with some real world usages, but we won’t need it for our simple examples. The sparse version is preferred when possible because it is faster to compute. Note that the final output of the model will always be one single class value, and this version is just to support a more flexible training data.

Let’s define our input method. We will reuse the `read_csv` function from the logistic regression example, but will call it with the defaults for the values on our dataset, which are all numeric.

```
def inputs():
    sepal_length, sepal_width, petal_length, petal_width, label = \
        read_csv(100, "iris.data", [[0.0], [0.0], [0.0], [0.0], [0.0]])
    # convert class names to a 0 based class index.
    label_number = tf.to_int32(tf.argmax(tf.to_int32(tf.pack([
        tf.equal(label, ["Iris-setosa"]),
        tf.equal(label, ["Iris-versicolor"]),
        tf.equal(label, ["Iris-virginica"])
    ])))
```

```

]), 0))

# Pack all the features that we care about in a single matrix;
# We then transpose to have a matrix with one example per row and one feature per column.
features = tf.transpose(tf.pack([sepal_length, sepal_width, petal_length, petal_width]))

return features, label_number

```

We don't need to convert each class to its own variable to use with `sparse_softmax_cross_entropy_with_logits`, but we need the value to be a number in the range of 0..2, since we have 3 possible classes. In the dataset file the class is a string value from the possible “Iris-setosa”, “Iris-versicolor”, or “Iris-virginica”. To convert it we create a tensor with `tf.pack`, comparing the file input with each possible value using `tf.equal`. Then we use `tf.argmax` to find the position on that tensor which is valued true, effectively converting the classes to a 0..2 integer.

The training function is also the same.

For evaluation of accuracy, we need a slight change from the sigmoid version:

```

def evaluate(sess, X, Y):
    predicted = tf.cast(tf.argmax(inference(X), 1), tf.int32)

    print sess.run(tf.reduce_mean(tf.cast(tf.equal(predicted, Y), tf.float32)))

```

The inference will compute the probabilities for each output class for our test examples. We use the `tf.argmax` function to choose the one with the highest probability as the predicted output value. Finally, we compare with the expected class with `tf.equal` and apply `tf.reduce_mean` just like with the sigmoid example.

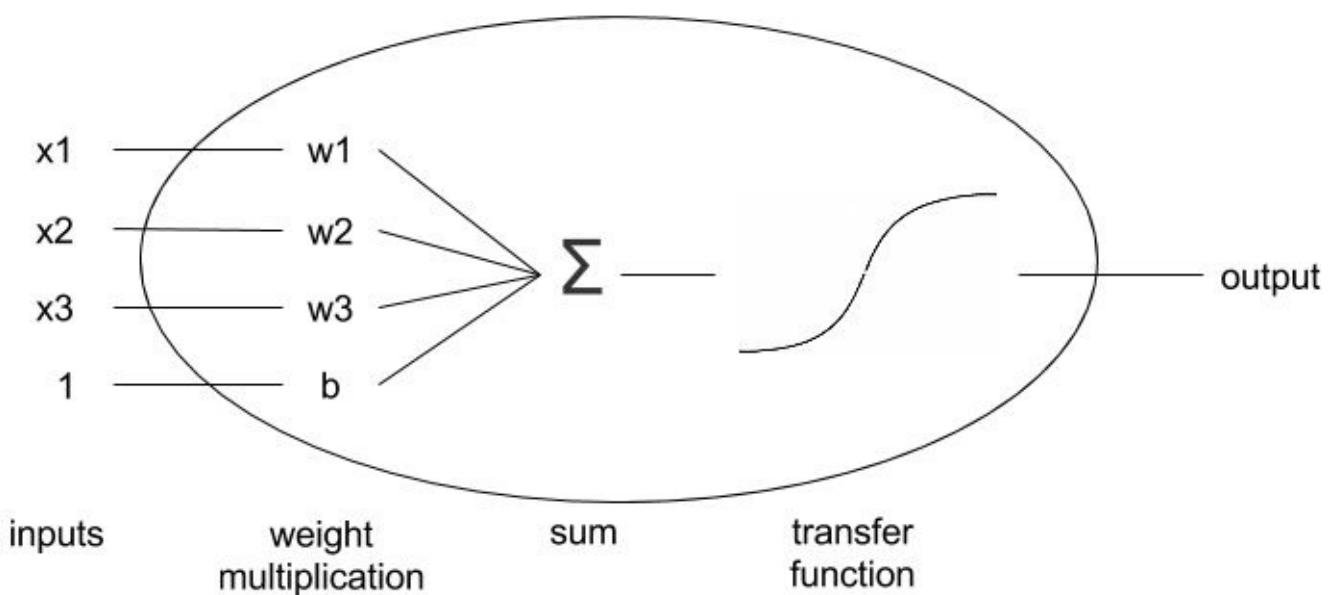
Running the code should print an accuracy of about 96%.

Multi-layer neural networks

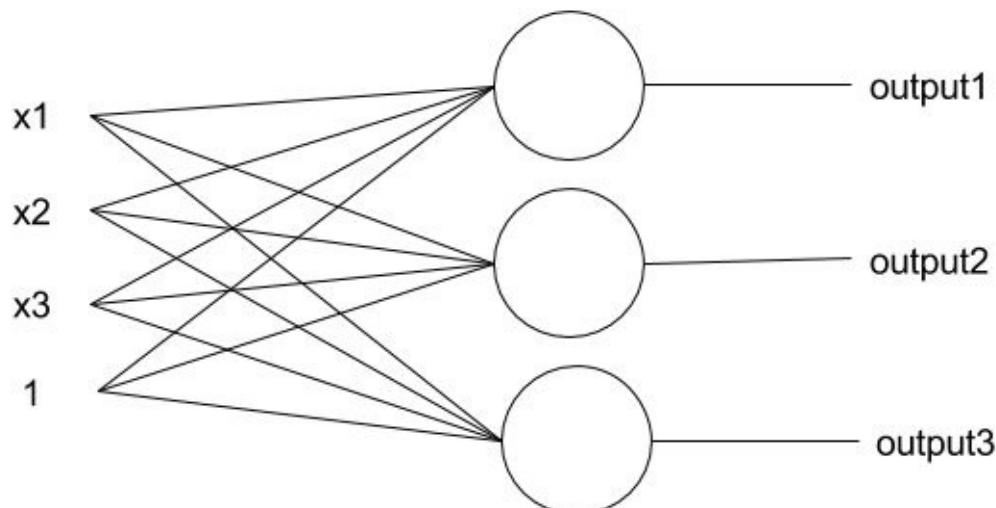
So far we have been using simple neural networks. Both linear and logistic regression models are single neurons that:

- Do a weighted sum of the input features. Bias can be considered the weight of an input feature that equals to 1 for every example. We call that doing a *linear combination* of the features.
- Then apply an **activation or transfer function** to calculate the output. In the case of the lineal regression, the transfer function is the identity (i.e. same value), while the logistic uses the sigmoid as the transfer.

The following diagram represents each neuron inputs, processing and output:



In the case of softmax classification, we used a network with C neurons- one for each possible output class:



Now, in order to resolve more difficult tasks, like reading handwritten digits, or actually

identifying cats and dogs on images, we are going to need a more developed model.

Lets start with a simple example. Suppose we want to build a network that learns how to fit the XOR (eXclusive OR) boolean operation:

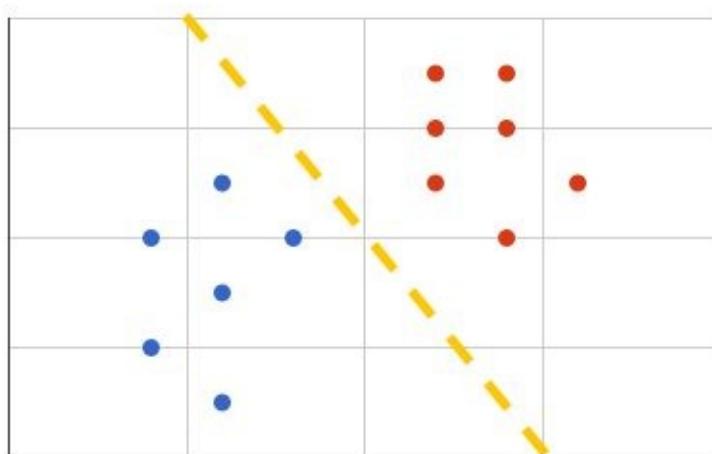
Table 4-1. XOR operation truth table

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

It should return 1 when either input equals to 1, but not when both do.

This seems to be a far more simpler problem than the ones we have tried so far, yet none of the models that we presented can solve it.

The reason is that sigmoid type of neurons require our data to be *linearly separable* to do their job well. That means that there must exist a straight line in 2 dimensional data (or hyperplane in higher dimensional data) that separates all the data examples belonging to a class in the same side of the plane, which looks something like this:



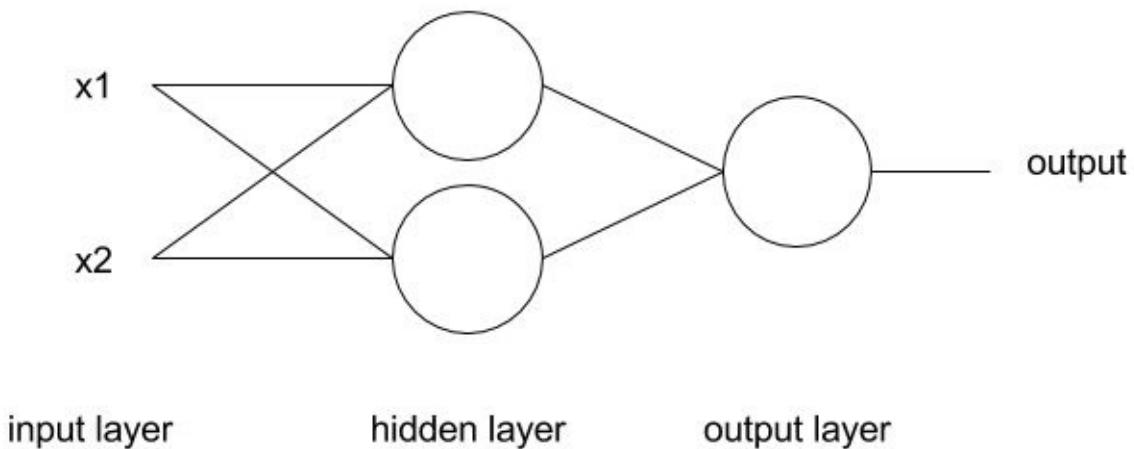
In the chart we can see example data samples as dots, with their associated class as the color. As long as we can find that yellow line completely separating the red and the blue dots in the chart, the sigmoid neuron will work fine for that dataset.

Let's look at the XOR gate function chart:



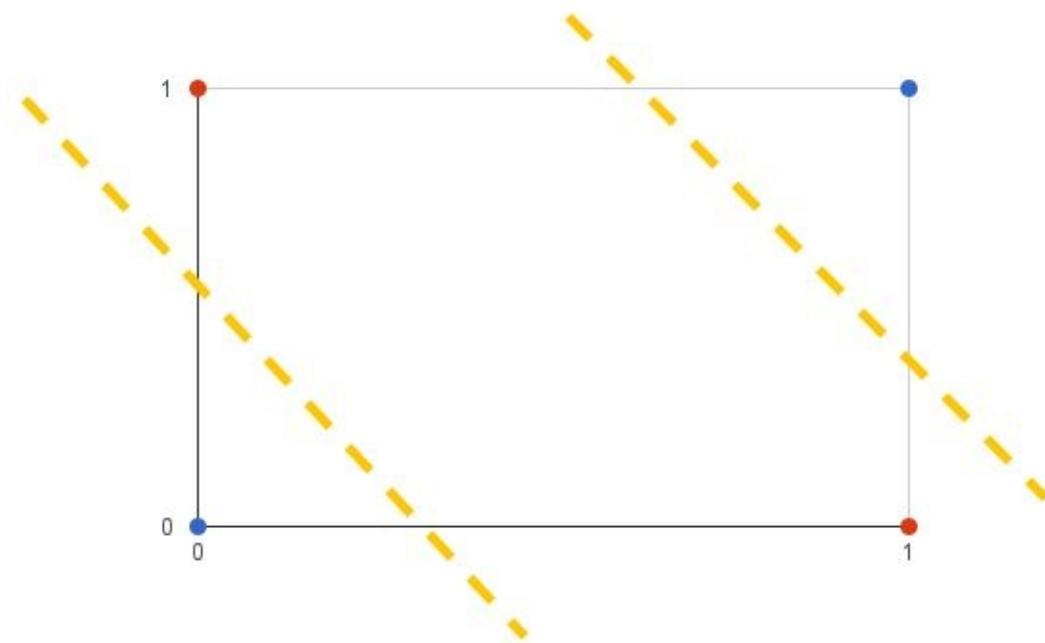
We can't find a single straight line that would split the chart, leaving all of the 1s (red dots) in one side and 0s (blue dots) in the other. That's because the XOR function output is not linearly separable.

This problem actually resulted in neural network research losing importance for about a decade around 1970's. So how did they fix the lack of linear separability to keep using networks? They did it by intercalating more neurons between the input and the output of the network, as you can see in the figure:



We say that we added a *hidden layer* of neurons between the input and the output layers. You can think of it as allowing our network to ask multiple questions to the input data, one question per neuron on the hidden layer, and finally deciding the output result based on the answers of those questions.

Graphically, we are allowing the network to draw more than one single separation line:



As you can see in the chart, each line divides the plane for the first questions asked to the input data. Then you can leave all of the equal outputs grouped together in a single area.

You can now guess what the *deep* means in deep learning. We make our networks deeper by adding more hidden layers on them. We may use different type of connections between them and even different activation functions in each layer.

Later in this book we present different types of deep neural networks for different usage scenarios.

Gradient descent and backpropagation

We cannot close the chapter about basic machine learning, without explaining how the learning algorithm we have been using works.

Gradient descent is an algorithm to find the points where a function achieves its minimum value. Remember that we defined learning as improving the model parameters in order to minimize the loss through a number of training steps. With that concept, applying gradient descent to find the minimum of the loss function will result in our model learning from our input data.

Let's define what a gradient is, in case you don't know. The gradient is a mathematical operation, generally represented with the ∇ symbol (nabla greek letter). It is analogous to a derivative, but applied to functions that input a vector and output a single value; like our loss functions do.

The output of the gradient is a vector of partial derivatives, one per position of the input vector of the function.

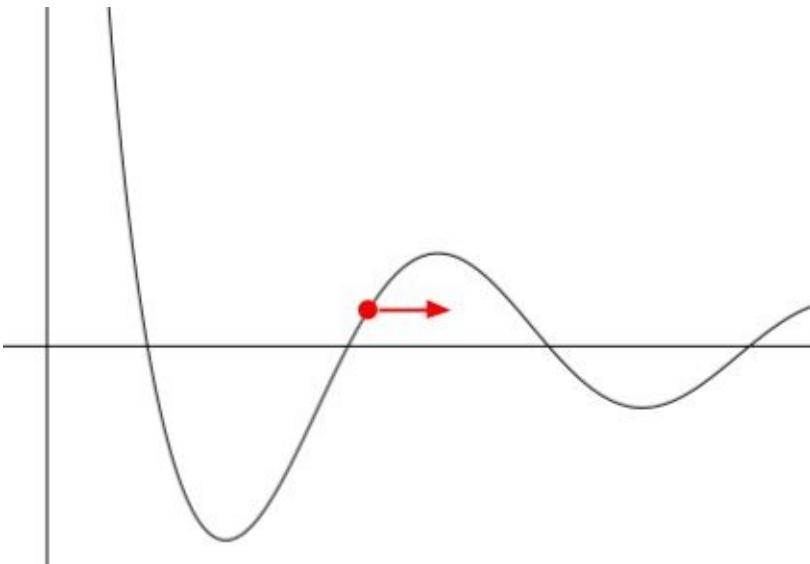
$$\nabla \equiv \left(\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \dots, \frac{\partial}{\partial w_N} \right)$$

You should think about a partial derivative as if your function would receive only one single variable, replacing all of the others by constants, and then applying the usual single variable derivation procedure.

The partial derivatives measure the rate of change of the function output with respect of a particular input variable. In other words, how much the output value will increase if we increase that input variable value.

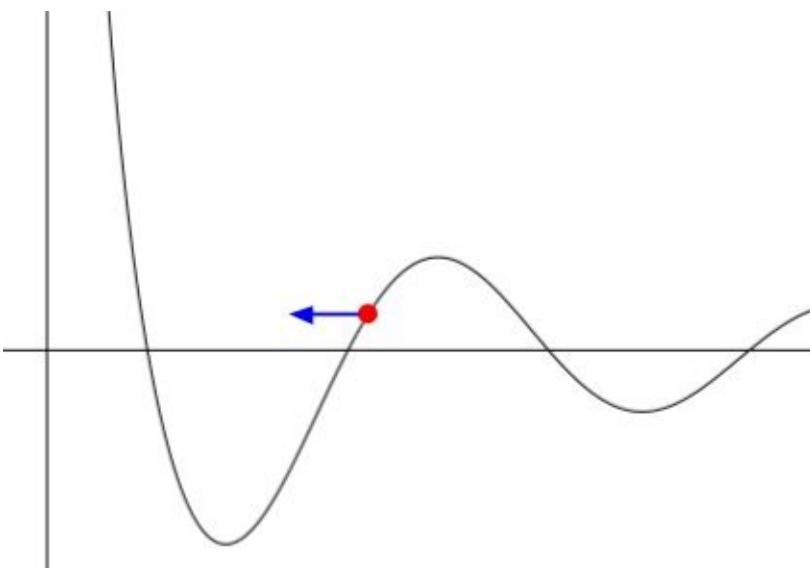
Here is a caveat before going on. When we talk about input variables of the loss function, we are referring to the model weights, not that actual dataset features inputs. Those are fixed by our dataset and cannot be optimized. The partial derivatives we calculate are with respect of each individual weight in the inference model.

We care about the gradient because its output vector indicates the direction of maximum growth for the loss function. You could think of it as a little arrow that will indicate in every point of the function where you should move to increase its value:



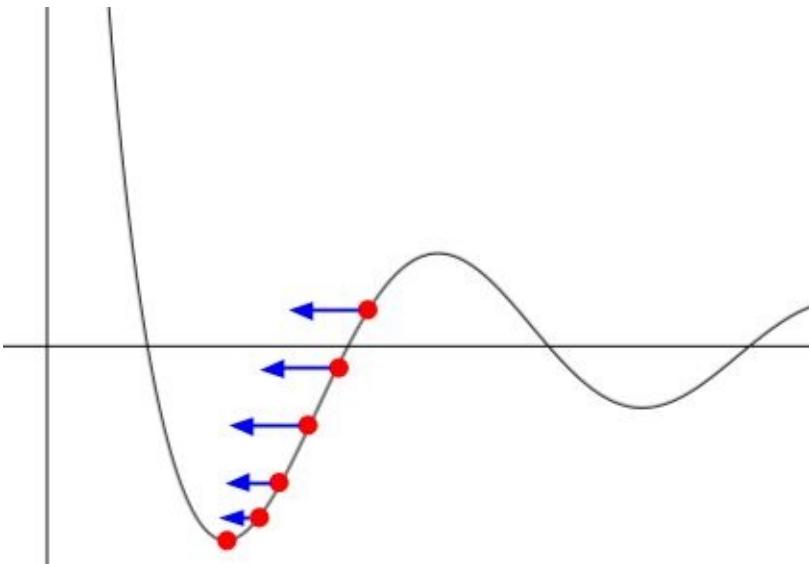
Suppose the chart above shows the loss function. The red dot represents the current weight values, where you are currently standing. The gradient represents the arrow, indicating that you should go right to increase the loss. Moreover, the length of the arrow indicates conceptually how much you would gain if you move in that direction.

Now, if we go in the opposite direction of the gradient, the loss will also do the opposite: decrease.



In the chart, if we go in the opposite direction of the gradient (blue arrow) we will go in the direction of decreasing loss.

If we move in that direction and calculate the gradient again, and then repeat the process until the gradient length is 0, we will arrive at the loss minimum. That is our goal, and graphically should look like:

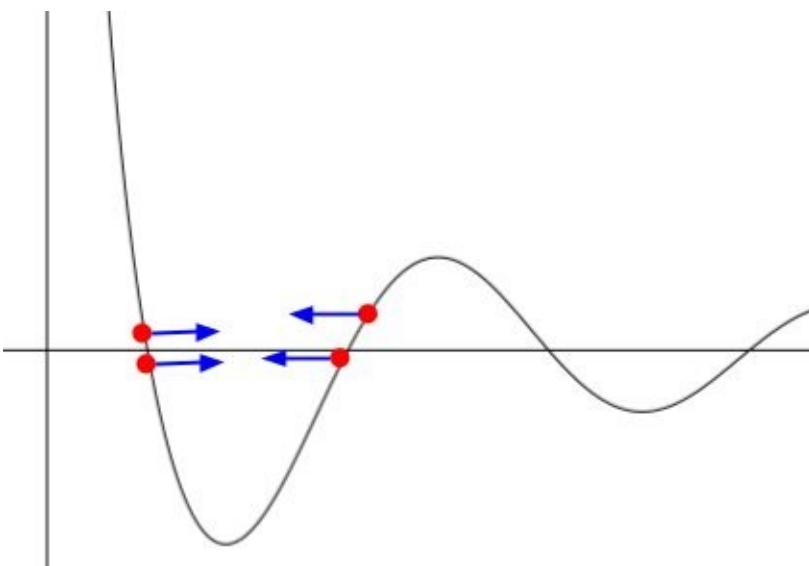


That's it. We can simply define gradient descent algorithm as:

$$weights_{step\ i+1} = weights_{step\ i} - \eta \cdot \nabla loss(weights_{step\ i})$$

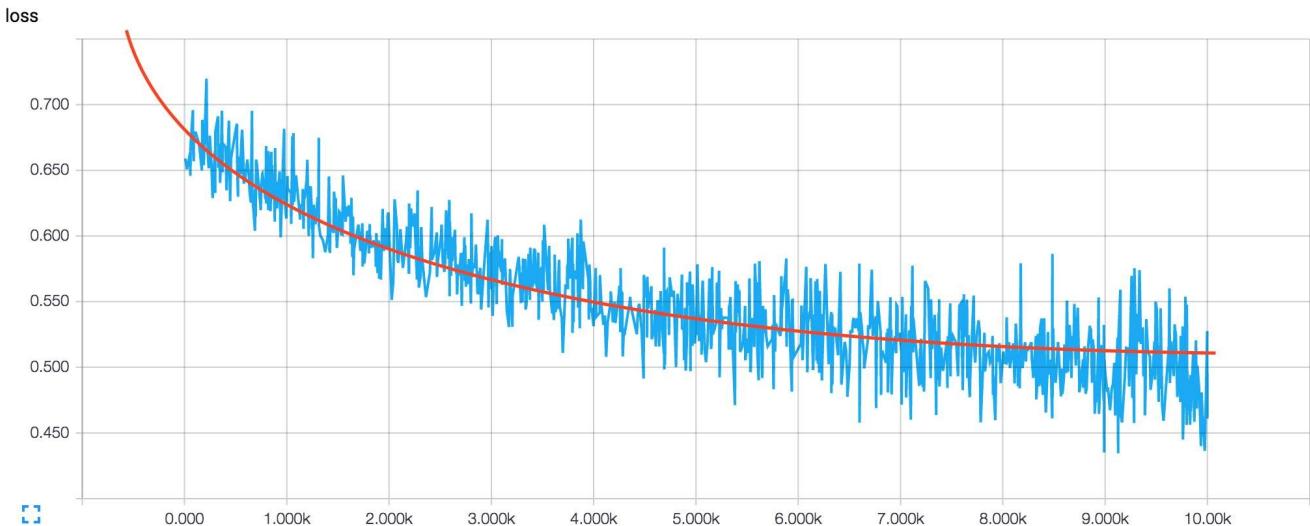
Notice how we added the η value to scale the gradient. We call it the learning rate. We need to add that because the length of the gradient vector is actually an amount measured in the “loss function units,” not in “weight units,” so we need to scale the gradient to be able to add it to our weights.

The learning rate is not a value that model will infer. It is an *hyperparameter*, or a manually configurable setting for our model. We need to figure out the right value for it. If it is too small then it will take many learning cycles to find the loss minimum. If it is too large, the algorithm may simply “skip over” the minimum and never find it, jumping cyclically. That's known as overshooting. In our example loss function chart, it would look like:



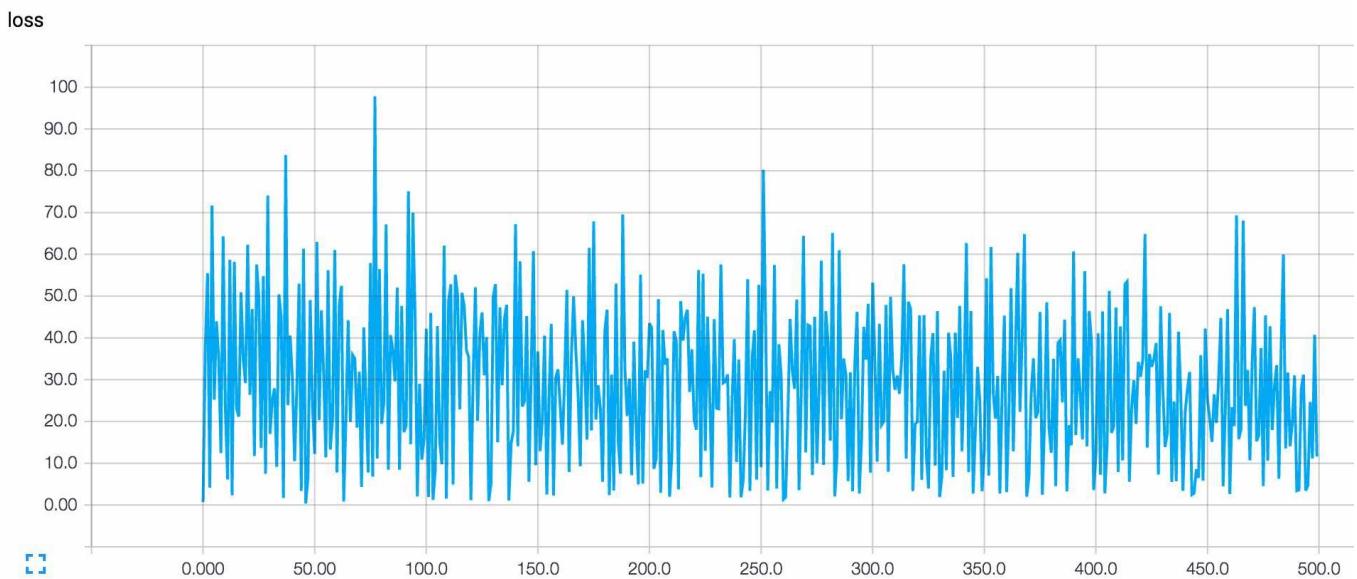
In practice, we can't really plot the loss function because it has many variables. So to know that we are trapped in overshooting, we have to look at the plot of the computed total loss thru time, which we can get in Tensorboard by using a `tf.scalar_summary` on the loss.

This is how a well behaving loss should diminish through time, indicating a good learning rate:



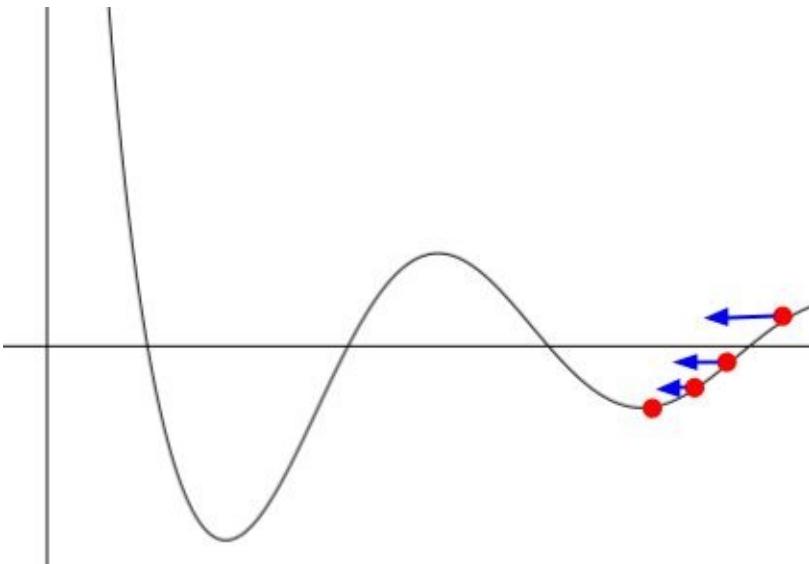
The blue line is the Tensorboard chart, and the red one represents the tendency line of the loss.

This is what it looks like when it is overshooting:



You should play with adjusting the learning rate so it is small enough that it doesn't overshoot, but is large enough to get it decaying quickly, so you can achieve learning faster using less cycles.

Besides the learning rate, other issues affect the gradient descent in the algorithm. The presence of local optima is in the loss function. Going back to the toy example loss function plot, this is how the algorithm would work if we had our initial weights close to the right side “valley” of the loss function:



The algorithm will find the valley and then stop because it will think that it is where the best possible value is located. The gradient is valued at 0 in all minima. The algorithm can't distinguish if it stopped in the absolute minimum of the function, the global minimum, or a local minimum that is the best value only in the close neighborhood.

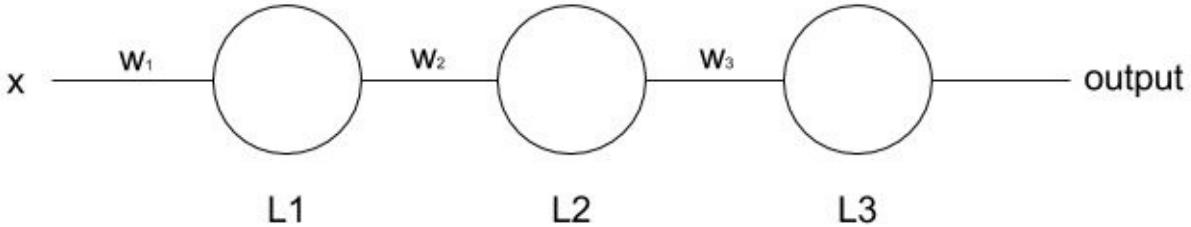
We try to fight against it by initializing the weights with random values. Remember that the first value for the weights is set manually. By using random values, we improve the chance to start descending closer from the global minimum.

In a deep network context like the ones we will see in later chapters, local minima are very frequent. A simple way to explain this is to think about how the same input can travel many different paths to the output, thus generating the same outcome. Luckily, there are papers showing that all of those minima are closely equivalent in terms of loss, and they are not really much worse than the global minimum.

So far we haven't been explicitly calculating any derivatives here, because we didn't have to. Tensorflow includes the method `tf.gradients` to symbolically compute the gradients of the specified graph steps and output that as tensors. We don't even need to manually call, because it also includes implementations of the gradient descent algorithm, among others. That is why we present high level formulas on how things should work without requiring us to go in-depth with implementation details and the math.

We are going to present through backpropagation. It is a technique used for efficiently computing the gradient in a computational graph.

Let's assume a really simple network, with one input, one output, and two hidden layers with a single neuron. Both hidden and output neurons will be sigmoids and the loss will be calculated using cross entropy. Such a network should look like:



Let's define $L1$ as the output of first hidden layer, $L2$ the output of the second, and $L3$ the final output of the network:

$$L1 = \text{sigmoid}(w_1 \cdot x)$$

$$L2 = \text{sigmoid}(w_2 \cdot L1)$$

$$L3 = \text{sigmoid}(w_3 \cdot L2)$$

Finally, the loss of the network will be:

$$\text{loss} = \text{cross_entropy}(L3, y_{\text{expected}})$$

To run one step of gradient decent, we need to calculate the partial derivatives of the loss function with respect of the three weights in the network. We will start from the output layer weights, applying the chain rule:

$$\frac{\partial \text{loss}}{\partial w_3} = \text{cross_entropy}'(L3, y_{\text{expected}}) \cdot \text{sigmoid}'(w_3 \cdot L2) \cdot L2$$

$L2$ is just a constant for this case as it doesn't depend on w_3

To simplify the expression we could define:

$$\text{loss}' = \text{cross_entropy}'(L3, y_{\text{expected}})$$

$$L3' = \text{sigmoid}'(w_3 \cdot L2)$$

The resulting expression for the partial derivative would be:

$$\frac{\partial \text{loss}}{\partial w_3} = \text{loss}' \cdot L3' \cdot L2$$

Now let's calculate the derivative for the second hidden layer weight, w_2 :

$$L2' = \text{sigmoid}'(w_2 \cdot L1)$$

$$\frac{\partial \text{loss}}{\partial w_2} = \text{loss}' \cdot L3' \cdot L2' \cdot L1$$

And finally the derivative for w_1 :

$$L1' = \text{sigmoid}'(w_1 \cdot x)$$

$$\frac{\partial \text{loss}}{\partial w_1} = \text{loss}' \cdot L3' \cdot L2' \cdot L1' \cdot x$$

You should notice a pattern. The derivative on each layer is the product of the derivatives of the layers after it by the output of the layer before. That's the magic of the chain rule and what the algorithm takes advantage of.

We go forward from the inputs calculating the outputs of each hidden layer up to the output layer. Then we start calculating derivatives going backwards through the hidden layers and *propagating* the results in order to do less calculations by reusing all of the elements already calculated. That's the origin of the name backpropagation.

Conclusion

Notice how we have not used the definition of the sigmoid or cross entropy derivatives. We could have used a network with different activation functions or loss and the result would be the same.

This is a very simple example, but in a network with thousands of weights to calculate their derivatives, using this algorithm can save orders of magnitude in training time.

To close, there are a few different optimization algorithms included in Tensorflow, though all of them are based in this method of computing gradients. Which one works better depends upon the shape of your input data and the problem you are trying to solve.

Sigmoid hidden layers, softmax output layers, and gradient descent with backpropagation are the most fundamentals blocks that we are going to use to build on for the more complex models that will see in the next chapters.

Part III. Implementing Advanced Deep Models in TensorFlow

Chapter 5. Object Recognition and Classification

At this point, you should have a basic understanding of TensorFlow and its best practices. We'll follow these practices while we build a model capable of object recognition and classification. Building this model expands on the fundamentals that have been covered so far while adding terms, techniques and fundamentals of computer vision. The technique used in training the model has become popular recently due to its accuracy across challenges.

[ImageNet](#), a database of labeled images, is where computer vision and deep learning saw a recent rise in popularity. Annually, ImageNet hosts a challenge (ILSVRC) where people build systems capable of automatically classifying and detecting objects based on ImageNet's database of images. In 2012, the challenge saw a team named [SuperVision](#) submit a solution using a creative neural network architecture. ILSVRC solutions are often creative but what set SuperVision's entry apart was its ability to accurately classify images. [SuperVision's entry](#) set a new standard for computer vision accuracy and stirred up interest in a deep learning technique named convolutional neural networks.

Convolutional Neural Networks (CNNs) have continued to grow in [popularity](#). They're primarily used for computer vision related tasks but are not limited to working with images. CNNs could be used with any data that can be represented as a tensor where values are ordered next to related values (in a grid). [Microsoft Research](#) released a paper in 2014 where they used CNNs for speech recognition where the input tensor was a single row grid of sound frequencies ordered by the time they were recorded. For images, the values in the tensor are pixels ordered in a grid corresponding with the width and height of the image.

In this chapter, the focus is working with CNNs and images in TensorFlow. The goal is to build a CNN model using TensorFlow that categorizes images based on a subset of ImageNet's database. Training a CNN model will require working with images in TensorFlow and understanding how convolutional neural networks (CNNs) are used. The majority of the chapter is dedicated to introducing concepts of computer vision using TensorFlow.

The dataset used in training this CNN model is a subset of the images available in ImageNet named the [Stanford's Dogs Dataset](#). As the name implies, this dataset is filled with images of different dog breeds and a label of the breed shown in the image. The goal of the model is to take an image and accurately guess the breed of dog shown in the image (example images are tagged as Siberian Husky from Stanford's Dog Databse).



n02110185_712.jpg

n02110185_4186.jpg

n02110185_1532.jpg

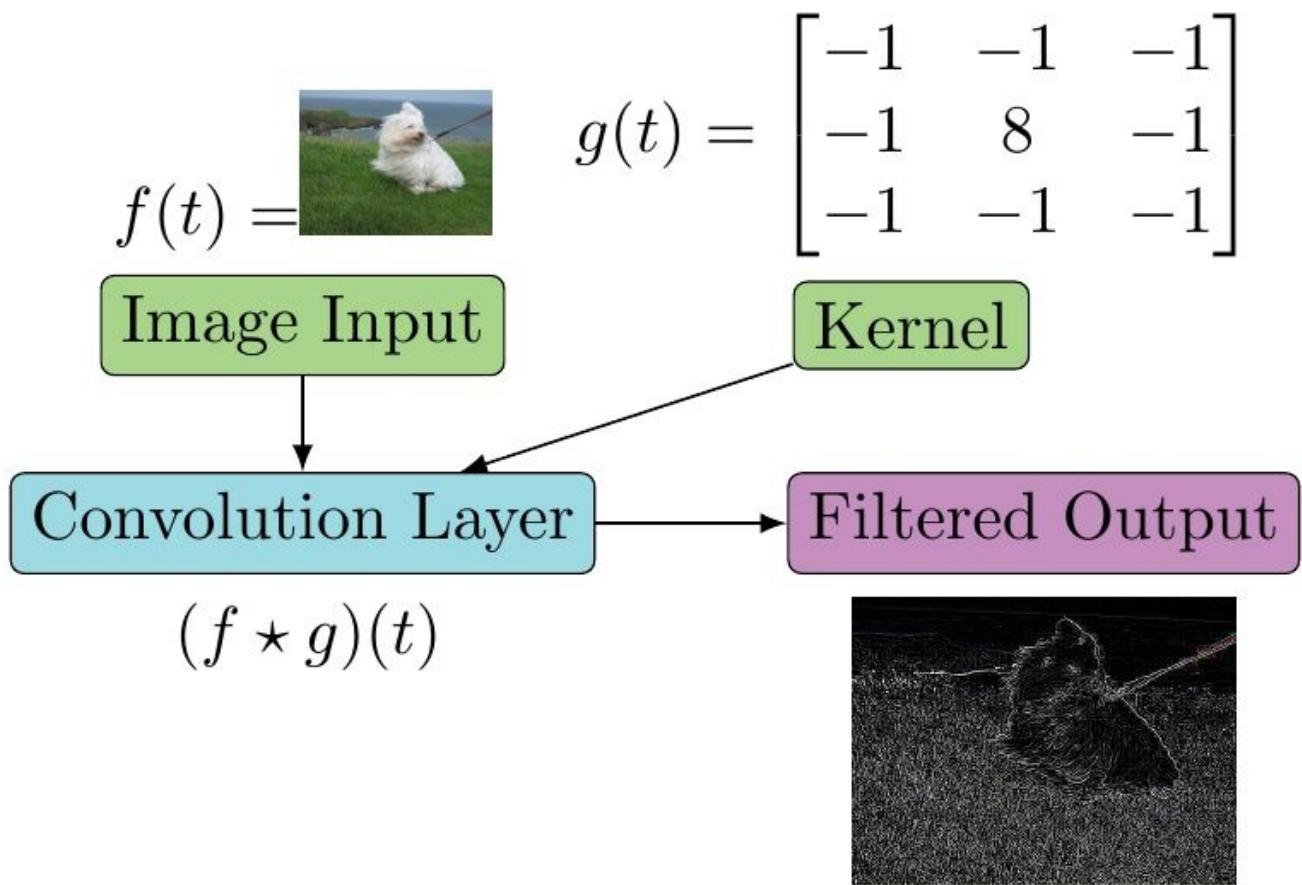
If one of the images shown above is loaded into the model, it should output a label of Siberian Husky. These example images wouldn't be a fair test of the model's accuracy because they exist in the training dataset. Finding a fair metric to calculate the model's accuracy requires a large number of images which won't be used in training. The images which haven't been used in training the model will be used to create a separate test dataset.

The reason to bring up the fairness of an image to test a model's accuracy is because it's part of keeping a separated test, train and cross-validation datasets. While processing input, it is a required practice to separate a large percentage of the data used to train a network. This separation is to allow a blind test of a model. Testing a model with input which was used to train it will likely create a model which accurately matches input it has already seen while not being capable of working with new input. The testing dataset is then used to see how well the model performs with data that didn't exist in the training. Over time and iterations of the model, it is possible that the changes being made to increase accuracy are making the model better equipped to the testing dataset while performing poorly in the real world. A good practice is to use a cross-validation dataset to check the final model and receive a better estimate of its accuracy. With images, it's best to separate the raw dataset while doing any preprocessing (color adjustments or cropping) keeping the input pipeline the same across all the datasets.

Convolutional Neural Networks

Technically, a convolutional neural network is a neural network which has at least one layer (`tf.nn.conv2d`) that does a convolution between its input f and a configurable kernel g generating the layer's output. In a simplified definition, a convolution's goal is to apply a kernel (filter) to every point in a tensor and generate a filtered output by sliding the kernel over an input tensor.

An example of the filtered output is edge detection in images. A special kernel is applied to each pixel of an image and the output is a new image depicting all the edges. In this case, the input tensor is an image and each point in the tensor is treated as a pixel which includes the amount of red, green and blue found at that point. The kernel is slid over every pixel in the image and the output value increases whenever there is an edge between colors. This figure shows the simplified convolution layer where the input is an image and the output is all the horizontal lines found in the image.



It isn't important to understand how convolutions combine input to generate filtered output, or what a kernel is, until later in this chapter when they're put in practice. Obtaining a broad sense of what a CNN does and its biological inspiration builds the technical implementation.

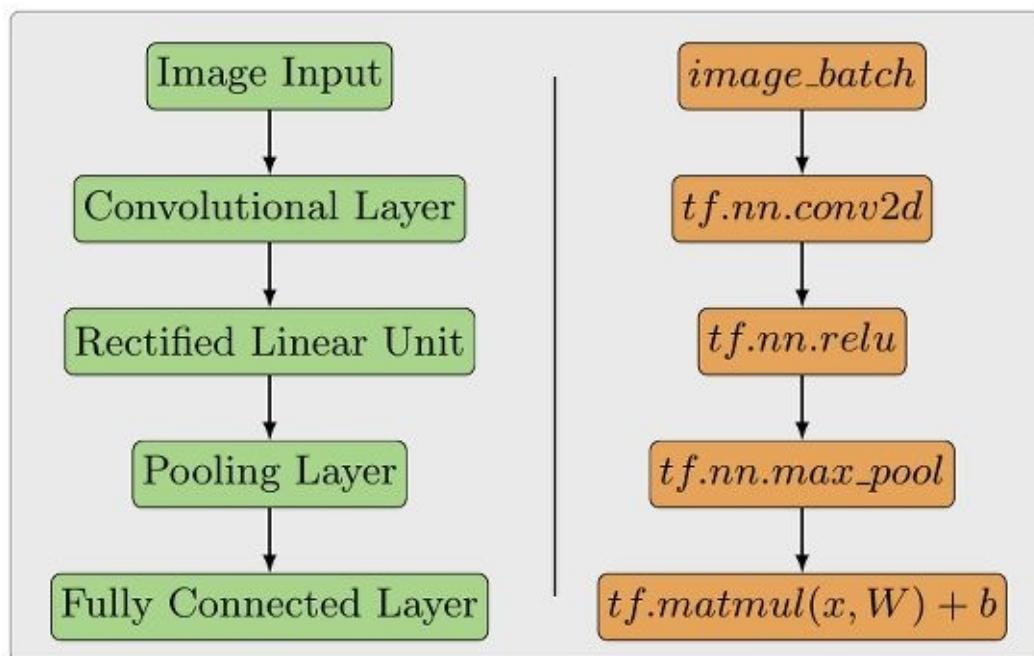
In 1968, [an article was published](#) detailing new findings on the cellular layout of a monkey striate cortex (the section of the brain thought to process visual input). The article

discusses a grouping of cells that extend vertically combining together to match certain visual traits. The study of primate brains may seem irrelevant to a machine learning task, yet it was instrumental [in the development of deep learning](#) using CNNs.

CNNs follow a simplified process matching information similar to the structure found in the cellular layout of a monkey's striate cortex. As signals are passed through a monkey's striate cortex, certain layers signal when a visual pattern is highlighted. For example, one layer of cells activate (increase its output signal) when a horizontal line passes through it. A CNN will exhibit a similar behavior where clusters of neurons will activate based on patterns learned from training. For example, after training, a CNN will have certain layers that activate when a horizontal line passes through it.

Matching horizontal lines would be a useful neural network architecture, but CNNs take it further by layering multiple simple patterns to match complex patterns. In the context of CNNs, these patterns are known as filters or kernels and the goal is to adjust these kernel weights until they accurately match the training data. Training these filters is often accomplished by combining multiple different layers and learning weights using gradient descent.

A simple CNN architecture may combine a convolutional layer (`tf.nn.conv2d`), non-linearity layer (`tf.nn.relu`), pooling layer (`tf.nn.max_pool`) and a fully connected layer (`tf.matmul`). Without these layers, it's difficult to match complex patterns because the network will be filled with too much information. A well designed CNN architecture highlights important information while ignoring noise. We'll go into details on how these layers work together later in this chapter.



The input image for this architecture is a complex format designed to support the ability to load batches of images. Loading a batch of images allows the computation of multiple images simultaneously but it requires a more complex data structure. The data structure used is a rank four tensor including all the information required to convolve a batch of

images. TensorFlow's input pipeline (which is used to read and decode files) has a special format designed to work with multiple images in a batch including required information for an image (`[image_batch_size, image_height, image_width, image_channels]`). Using the example code, it's possible to examine the structure of an example input used while working with images in TensorFlow.

```
image_batch = tf.constant([
    [ # First Image
        [[0, 255, 0], [0, 255, 0], [0, 255, 0]],
        [[0, 255, 0], [0, 255, 0], [0, 255, 0]]
    ],
    [ # Second Image
        [[0, 0, 255], [0, 0, 255], [0, 0, 255]],
        [[0, 0, 255], [0, 0, 255], [0, 0, 255]]
    ]
])
image_batch.get_shape()
```

The output from executing the example code is:

```
TensorShape([Dimension(2), Dimension(2), Dimension(3), Dimension(3)])
```

NOTE: The example code and further examples in this chapter do not include the common bootstrapping required to run TensorFlow code. This includes importing the `tensorflow` (usually as `tf` for brevity), creating a TensorFlow session as `sess`, initializing all variables, and starting thread runners. Undefined variable errors may occur if the example code is executed without running these steps.

In this example code, a batch of images is created that includes two images. Each image has a height of two pixels and a width of three pixels with an RGB color space. The output from executing the example code shows the amount of images as the size of the first set of dimensions `Dimension(2)`, the height of each image as the size of the second set `Dimension(2)`, the width of each image as the third set `Dimension(3)`, and the size of the color channel as the final set `Dimension(3)`.

It's important to note each pixel maps to the height and width of the image. Retrieving the first pixel of the first image requires each dimension accessed as follows.

```
sess.run(image_batch)[0][0][0]
```

The output from executing the example code is:

```
array([ 0, 255, 0], dtype=int32)
```

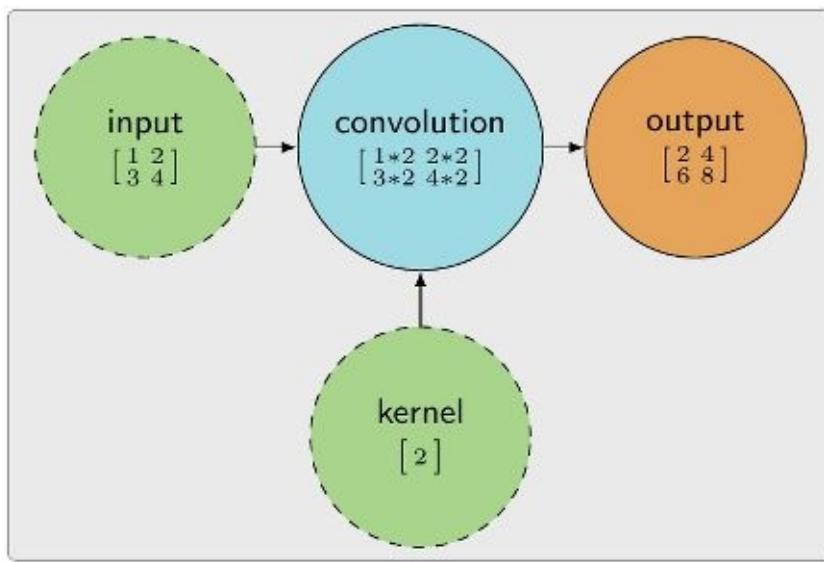
Instead of loading images from disk, the `image_batch` variable will act as if it were images loaded as part of an input pipeline. Images loaded from disk using an input pipeline have the same format and act the same. It's often useful to create fake data similar to the `image_batch` example above to test input and output from a CNN. The simplified input will make it easier to debug any simple issues. It's important to work on simplification of debugging because CNN architectures are incredibly complex and often take days to train.

The first complexity working with CNN architectures is how a convolution layer works. After any image loading and manipulation, a convolution layer is often the first layer in the network. The first convolution layer is useful because it can simplify the rest of the network and be used for debugging. The next section will focus on how convolution layers

operate and using them with TensorFlow.

Convolution

As the name implies, convolution operations are an important component of convolutional neural networks. The ability for a CNN to accurately match diverse patterns can be attributed to using convolution operations. These operations require complex input, which was shown in the previous section. In this section we'll experiment with convolution operations and the parameters that are available to tune them. Here the convolution operation convolves two input tensors (input and kernel) into a single output tensor, which represents information from each input.



Input and Kernel

Convolution operations in TensorFlow are done using `tf.nn.conv2d` in a typical situation. There are other convolution operations available using TensorFlow designed with special use cases. `tf.nn.conv2d` is the preferred convolution operation to begin experimenting with. For example, we can experiment with convolving two tensors together and inspect the result.

```
input_batch = tf.constant([
    [ # First Input
        [[0.0], [1.0]],
        [[2.0], [3.0]]
    ],
    [ # Second Input
        [[2.0], [4.0]],
        [[6.0], [8.0]]
    ]
])

kernel = tf.constant([
    [
        [[1.0, 2.0]]
    ]
])
```

The example code creates two tensors. The `input_batch` tensor has a similar shape to the `image_batch` tensor seen in the previous section. This will be the first tensor being convolved and the second tensor will be `kernel`. *Kernel* is an important term that is interchangeable with *weights*, *filter*, *convolution matrix* or *mask*. Since this task is computer vision related, it's useful to use the term `kernel` because it is being treated as an [image kernel](#). There is no practical difference in the term when used to describe this functionality in TensorFlow. The parameter in TensorFlow is named `filter` and it expects a set of weights which will be learned from training. The amount of different weights included in the kernel (`filter` parameter) will configure the amount of kernels that will be learned.

In the example code, there is a single kernel which is the first dimension of the `kernel` variable. The kernel is built to return a tensor that will include one channel with the original input and a second channel with the original input doubled. In this case, channel is used to describe the elements in a rank 1 tensor (vector). Channel is a term from computer vision that describes the output vector, for example an RGB image has three channels represented as a rank 1 tensor [red, green, blue]. At this time, ignore the `strides` and `padding` parameter, which will be covered later, and focus on the convolution (`tf.nn.conv2d`) output.

```
conv2d = tf.nn.conv2d(input_batch, kernel, strides=[1, 1, 1, 1], padding='SAME')
sess.run(conv2d)
```

The output from executing the example code is:

```
array([[[[ 0.,  0.],
        [ 1.,  2.]],
       [[ 2.,  4.],
        [ 3.,  6.]]],
      [[[ 2.,  4.],
        [ 4.,  8.]],
       [[ 6., 12.],
        [ 8., 16.]]]], dtype=float32)
```

The output is another tensor which is the same rank as the `input_batch` but includes the

number of dimensions found in the kernel. Consider if `input_batch` represented an image, the image would have a single channel, in this case it could be considered a grayscale image. Each element in the tensor would represent one pixel of the image. The pixel in the bottom right corner of the image would have the value of `3.0`.

Consider the `tf.nn.conv2d` convolution operation as a combination of the image (represented as `input_batch`) and the `kernel` tensor. The convolution of these two tensors create a feature map. Feature map is a broad term except in computer vision where it relates to the output of operations which work with an image kernel. The feature map now represents the convolution of these tensors by adding new layers to the output.

The relationship between the input images and the output feature map can be explored with code. Accessing elements from the input batch and the feature map are done using the same index. By accessing the same pixel in both the input and the feature map shows how the input was changed when it convolved with the `kernel`. In the following case, the lower right pixel in the image was changed to output the value found by multiplying `3.0 * 1.0` and `3.0 * 2.0`. The values correspond to the pixel value and the corresponding value found in the `kernel`.

```
lower_right_image_pixel = sess.run(input_batch)[0][1][1]
lower_right_kernel_pixel = sess.run(conv2d)[0][1][1]

lower_right_image_pixel, lower_right_kernel_pixel
```

The output from executing the example code is:

```
(array([ 3.], dtype=float32), array([ 3.,  6.], dtype=float32))
```

In this simplified example, each pixel of every image is multiplied by the corresponding value found in the kernel and then added to a corresponding layer in the feature map. Layer, in this context, is referencing a new dimension in the output. With this example, it's hard to see a value in convolution operations.

Strides

The value of convolutions in computer vision is their ability to reduce the dimensionality of the input, which is an image in this case. An image's dimensionality (2D image) is its width, height and number of channels. A large image dimensionality requires an exponentially larger amount of time for a neural network to scan over every pixel and judge which ones are important. Reducing dimensionality of an image with convolutions is done by altering the `strides` of the kernel.

The parameter `strides`, causes a kernel to skip over pixels of an image and not include them in the output. It's not fair to say the pixels are skipped because they still may affect the output. The `strides` parameter highlights how a convolution operation is working with a kernel when a larger image and more complex kernel are used. As a convolution is sliding the kernel over the input, it's using the `strides` parameter to change how it walks over the input. Instead of going over every element of an input, the `strides` parameter could configure the convolution to skip certain elements.

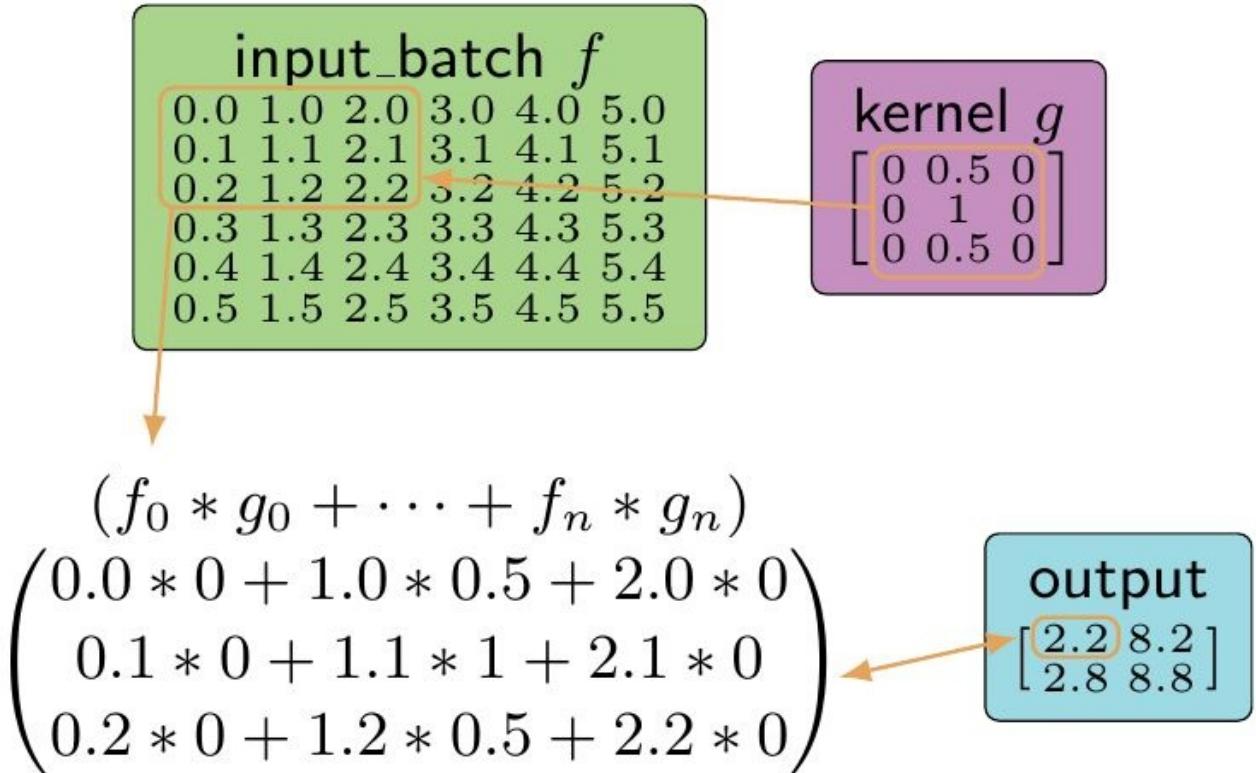
For example, take the convolution of a larger image and a larger kernel. In this case, it's a convolution between a 6 pixel tall, 6 pixel wide and 1 channel deep image (6x6x1) and a (3x3x1) kernel.

```
input_batch = tf.constant([
    # First Input (6x6x1)
    [[[0.0], [1.0], [2.0], [3.0], [4.0], [5.0]],
     [[0.1], [1.1], [2.1], [3.1], [4.1], [5.1]],
     [[0.2], [1.2], [2.2], [3.2], [4.2], [5.2]],
     [[0.3], [1.3], [2.3], [3.3], [4.3], [5.3]],
     [[0.4], [1.4], [2.4], [3.4], [4.4], [5.4]],
     [[0.5], [1.5], [2.5], [3.5], [4.5], [5.5]],
    ],
])
kernel = tf.constant([
    # Kernel (3x3x1)
    [[[0.0]], [[0.5]], [[0.0]]],
    [[[0.0]], [[1.0]], [[0.0]]],
    [[[0.0]], [[0.5]], [[0.0]]]
])
# NOTE: the change in the size of the strides parameter.
conv2d = tf.nn.conv2d(input_batch, kernel, strides=[1, 3, 3, 1], padding='SAME')
sess.run(conv2d)
```

The output from executing the example code is:

```
array([[[[ 2.20000005,
          [ 8.19999981]],
         [[ 2.79999995,
          [ 8.80000019]]]], dtype=float32)
```

The `input_batch` was combined with the `kernel` by moving the `kernel` over the `input_batch` striding (or skipping) over certain elements. Each time the `kernel` was moved, it get centered over an element of `input_batch`. Then the overlapping values are multiplied together and the result is added together. This is how a convolution combines two inputs using what's referred to as pointwise multiplication. It may be easier to visualize using the following figure.



In this figure, the same logic follows what is found in the code. Two tensors convolved together while striding over the input. The strides reduced the dimensionality of the output a large amount while the kernel size allowed the convolution to use all the input values. None of the input data was completely removed from striding but now the input is a smaller tensor.

Strides are a way to adjust the dimensionality of input tensors. Reducing dimensionality requires less processing power, and will keep from creating receptive fields which completely overlap. The `strides` parameter follows the same format as the input tensor `[image_batch_size_stride, image_height_stride, image_width_stride, image_channels_stride]`. Changing the first or last element of the stride parameter are rare, they'd skip data in a `tf.nn.conv2d` operation and not take the input into account. The `image_height_stride` and `image_width_stride` are useful to alter in reducing input dimensionality.

A challenge that comes up often with striding over the input is how to deal with a stride which doesn't evenly end at the edge of the input. The uneven striding will come up often due to image size and kernel size not matching the striding. If the image size, kernel size and strides can't be changed then padding can be added to the image to deal with the uneven area.

Padding

When a kernel is overlapped on an image it should be set to fit within the bounds of the image. At times, the sizing may not fit and a good alternative is to fill the missing area in the image. Filling the missing area of the image is known as padding the image.

TensorFlow will pad the image with zeros or raise an error when the sizes don't allow a kernel to stride over an image without going past its bounds. The amount of zeros or the error state of `tf.nn.conv2d` is controlled by the parameter `padding` which has two possible values ('VALID', 'SAME').

SAME: The convolution output is the **SAME** size as the input. This doesn't take the filter's size into account when calculating how to stride over the image. This may stride over more of the image than what exists in the bounds while padding all the missing values with zero.

VALID: Take the filter's size into account when calculating how to stride over the image. This will try to keep as much of the kernel inside the image's bounds as possible. There may be padding in some cases but will avoid.

It's best to consider the size of the input but if padding is necessary then TensorFlow has the option built in. In most simple scenarios, `SAME` is a good choice to begin with. `VALID` is preferential when the input and kernel work well with the strides. For further information, TensorFlow covers this subject well in the [convolution documentation](#).

Data Format

There's another parameter to `tf.nn.conv2d` which isn't shown from these examples named `data_format`. The [tf.nn.conv2d docs](#) explain how to change the data format so the `input`, `kernel` and `strides` follow a format other than the format being used thus far. Changing this format is useful if there is an input tensor which doesn't follow the `[batch_size, height, width, channel]` standard. Instead of changing the input to match, it's possible to change the `data_format` parameter to use a different layout.

`data_format`: An optional string from: “NHWC”, “NCHW”. Defaults to “NHWC”. Specify the data format of the input and output data. With the default format “NHWC”, the data is stored in the order of: `[batch, in_height, in_width, in_channels]`. Alternatively, the format could be “NCHW”, the data storage order of: `[batch, in_channels, in_height, in_width]`.

Data Format	Definition
N	Number of tensors in a batch, the <code>batch_size</code> .
H	Height of the tensors in each batch.
W	Width of the tensors in each batch.
C	Channels of the tensors in each batch.

Kernels in Depth

In TensorFlow the filter parameter is used to specify the kernel convolved with the input. Filters are commonly used in photography to adjust attributes of a picture, such as the amount of sunlight allowed to reach a camera's lens. In photography, filters allow a photographer to drastically alter the picture they're taking. The reason the photographer is able to alter their picture using a filter is because the filter can recognize certain attributes of the light coming in to the lens. For example, a red lens filter will absorb (block) every frequency of light which isn't red allowing only red to pass through the filter.



Before and after applying a minor red filter to n02088466_3184.jpg.

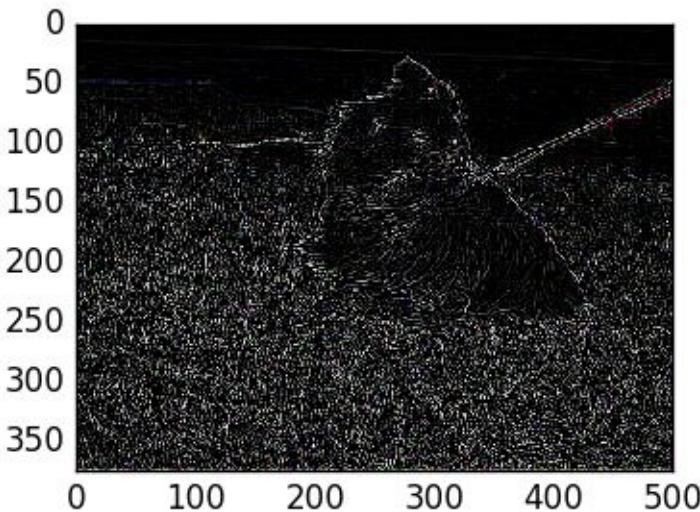
In computer vision, kernels (filters) are used to recognize important attributes of a digital image. They do this by using certain patterns to highlight when features exist in an image. A kernel which will replicate the red filter example image is implemented by using a reduced value for all colors except red. In this case, the reds will stay the same but all other colors matched are reduced.

The example seen at the start of this chapter uses a kernel designed to do edge detection. Edge detection kernels are common in computer vision applications and could be implemented using basic TensorFlow operations and a single `tf.nn.conv2d` operation.

```
kernel = tf.constant([
  [
    [[ -1.,  0.,  0.], [ 0., -1.,  0.], [ 0.,  0., -1.]],
    [[ -1.,  0.,  0.], [ 0., -1.,  0.], [ 0.,  0., -1.]],
    [[ -1.,  0.,  0.], [ 0., -1.,  0.], [ 0.,  0., -1.]]
  ],
  [
    [[ -1.,  0.,  0.], [ 0., -1.,  0.], [ 0.,  0., -1.]],
    [[ 8.,  0.,  0.], [ 0.,  8.,  0.], [ 0.,  0.,  8.]],
    [[ -1.,  0.,  0.], [ 0., -1.,  0.], [ 0.,  0., -1.]]
  ],
  [
    [[ -1.,  0.,  0.], [ 0., -1.,  0.], [ 0.,  0., -1.]],
    [[ -1.,  0.,  0.], [ 0., -1.,  0.], [ 0.,  0., -1.]],
    [[ -1.,  0.,  0.], [ 0., -1.,  0.], [ 0.,  0., -1.]]
  ]
])
```



```
conv2d = tf.nn.conv2d(image_batch, kernel, [1, 1, 1, 1], padding="SAME")
activation_map = sess.run(tf.minimum(tf.nn.relu(conv2d), 255))
```



The output created from convolving an image with an edge detection kernel are all the areas where an edge was detected. The code assumes a batch of images is already available (`image_batch`) with a real image loaded from disk. In this case, the image is an example image found in the Stanford Dogs Dataset. The kernel has three input and three output channels. The channels sync up to RGB values between $[0, 255]$ with 255 being the maximum intensity. The `tf.minimum` and `tf.nn.relu` calls are there to keep the convolution values within the range of valid RGB colors of $[0, 255]$.

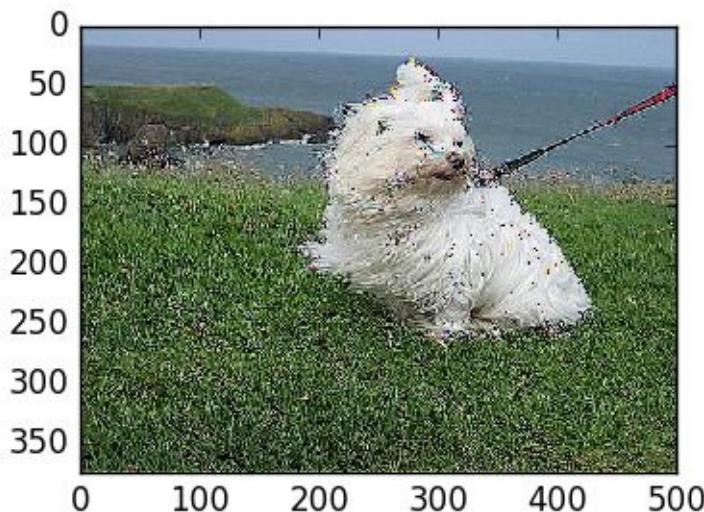
There are [many other](#) common kernels which can be used in this simplified example. Each will highlight different patterns in an image with different results. The following kernel will sharpen an image by increasing the intensity of color changes.

```

kernel = tf.constant([
  [
    [[ 0.,  0.,  0.], [ 0.,  0.,  0.], [ 0.,  0.,  0.]],
    [[ -1.,  0.,  0.], [ 0., -1.,  0.], [ 0.,  0., -1.]],
    [[ 0.,  0.,  0.], [ 0.,  0.,  0.], [ 0.,  0.,  0.]]
  ],
  [
    [[ -1.,  0.,  0.], [ 0., -1.,  0.], [ 0.,  0., -1.]],
    [[ 5.,  0.,  0.], [ 0.,  5.,  0.], [ 0.,  0.,  5.]],
    [[ -1.,  0.,  0.], [ 0., -1.,  0.], [ 0.,  0., -1.]]
  ],
  [
    [[ 0.,  0.,  0.], [ 0.,  0.,  0.], [ 0.,  0.,  0.]],
    [[ -1.,  0.,  0.], [ 0., -1.,  0.], [ 0.,  0., -1.]],
    [[ 0.,  0.,  0.], [ 0.,  0.,  0.], [ 0.,  0.,  0.]]
  ]
])

conv2d = tf.nn.conv2d(image_batch, kernel, [1, 1, 1, 1], padding="SAME")
activation_map = sess.run(tf.minimum(tf.nn.relu(conv2d), 255))

```



The values in the kernel were adjusted with the center of the kernel increased in intensity and the areas around the kernel reduced in intensity. The change, matches patterns with intense pixels and increases their intensity outputting an image which is visually sharpened. Note that the corners of the kernel are all 0 and don't affect the output that operates in a plus shaped pattern.

These kernels match patterns in images at a rudimentary level. A convolutional neural network matches edges and more by using a complex kernel it learned during training. The starting values for the kernel are usually random and over time they're trained by the CNN's learning layer. When a CNN is complete, it starts running and each image sent in is convolved with a kernel which is then changed based on if the predicted value matches the labeled value of the image. For example, if a Sheepdog picture is considered a Pit Bull by the CNN being trained it will then change the filters a small amount to try and match Sheepdog pictures better.

Learning complex patterns with a CNN involves more than a single layer of convolution. Even the example code included a `tf.nn.relu` layer used to prepare the output for visualization. Convolution layers may occur more than once in a CNN but they'll likely include other layer types as well. These layers combined form the support network required for a successful CNN architecture.

Common Layers

For a neural network architecture to be considered a CNN, it requires at least one convolution layer (`tf.nn.conv2d`). There are practical uses for a single layer CNN (edge detection), for image recognition and categorization it is common to use different layer types to support a convolution layer. These layers help reduce over-fitting, speed up training and decrease memory usage.

The layers covered in this chapter are focused on layers commonly used in a CNN architecture. A CNN isn't limited to use only these layers, they can be mixed with layers designed for other network architectures.

Convolution Layers

One type of convolution layer has been covered in detail (`tf.nn.conv2d`) but there are a few notes which are useful to advanced users. The convolution layers in TensorFlow don't do a full convolution, details can be found in [the TensorFlow API documentation](#). In practice, the difference between a convolution and the operation TensorFlow uses is performance. TensorFlow uses a technique to speed up the convolution operation in all the different types of convolution layers.

There are use cases for each type of convolution layer but for `tf.nn.conv2d` is a good place to start. The other types of convolutions are useful but not required in building a network capable of object recognition and classification. A brief summary of each is included.

`tf.nn.depthwise_conv2d`

This convolution is used when attaching the output of one convolution to the input of another convolution layer. An advanced use case is using a `tf.nn.depthwise_conv2d` to create a network following the [inception architecture](#).

`tf.nn.separable_conv2d`

This is similar to `tf.nn.conv2d`, but not a replacement for it. For large models, it speeds up training without sacrificing accuracy. For small models, it will converge quickly with worse accuracy.

`tf.nn.conv2d_transpose`

This applies a kernel to a new feature map where each section is filled with the same values as the kernel. As the kernel strides over the new image, any overlapping sections are summed together. There is a great explanation on how `tf.nn.conv2d_transpose` is used for learnable upsampling in [Stanford's CS231n Winter 2016: Lecture 13](#).

Activation Functions

These functions are used in combination with the output of other layers to generate a feature map. They're used to smooth (or differentiate) the results of certain operations. The goal is to introduce non-linearity into the neural network. Non-linearity means that the input is a curve instead of a straight line. Curves are capable of representing more complex changes in input. For example, non-linear input is capable of describing input which stays small for the majority of the time but periodically has a single point at an extreme. Introduction of non-linearity in a neural network allows it to train on the complex patterns found in data.

TensorFlow has [multiple activation functions](#) available. With CNNs, `tf.nn.relu` is primarily used because of its performance although it sacrifices information. When starting out, using `tf.nn.relu` is recommended but advanced users may create their own. When considering if an activation function is useful there are a few primary considerations.

1. The function is [monotonic](#), so its output should always be increasing or decreasing along with the input. This allows gradient descent optimization to search for local minima.
2. The function is [differentiable](#), so there must be a derivative at any point in the function's domain. This allows gradient descent optimization to properly work using the output from this style of activation function.

Any functions that satisfy those considerations could be used as activation functions. In TensorFlow there are a few worth highlighting which are common to see in CNN architectures. A brief summary of each is included with a small sample code illustrating their usage.

`tf.nn.relu`

A rectifier (rectified linear unit) called a ramp function in some documentation and looks like a skateboard ramp when plotted. ReLU is linear and keeps the same input values for any positive numbers while setting all negative numbers to be 0. It has the benefit that it doesn't suffer from [gradient vanishing](#) and has a range of $[0, +\infty)$. A drawback of ReLU is that it can suffer from neurons becoming saturated when too high of a learning rate is used.

```
features = tf.range(-2, 3)
# Keep note of the value for negative features
sess.run([features, tf.nn.relu(features)])
```

The output from executing the example code is:

```
[array([-2, -1, 0, 1, 2], dtype=int32), array([0, 0, 0, 1, 2], dtype=int32)]
```

In this example, the input is a rank one tensor (vector) of integer values between $[-2, 3]$. A `tf.nn.relu` is run over the values the output highlights that any value less than 0 is set to be 0. The other input values are left untouched.

tf.sigmoid

A sigmoid function returns a value in the range of $[0.0, 1.0]$. Larger values sent into a `tf.sigmoid` will trend closer to 1.0 while smaller values will trend towards 0.0. The ability for sigmoids to keep values between $[0.0, 1.0]$ is useful in networks which train on probabilities which are in the range of $[0.0, 1.0]$. The reduced range of output values can cause trouble with input becoming saturated and changes in input becoming exaggerated.

```
# Note, tf.sigmoid (tf.nn.sigmoid) is currently limited to float values
features = tf.to_float(tf.range(-1, 3))
sess.run([features, tf.sigmoid(features)])
```

The output from executing the example code is:

```
[array([-1., 0., 1., 2.], dtype=float32),
 array([ 0.26894143, 0.5, 0.7310586, 0.88079703], dtype=float32)]
```

In this example, a range of integers is converted to be float values (1 becomes 1.0) and a sigmoid function is ran over the input features. The result highlights that when a value of 0.0 is passed through a sigmoid, the result is 0.5 which is the midpoint of the sigmoid's domain. It's useful to note that with 0.5 being the sigmoid's midpoint, negative values can be used as input to a sigmoid.

tf.tanh

A hyperbolic tangent function (tanh) is a close relative to `tf.sigmoid` with some of the same benefits and drawbacks. The main difference between `tf.sigmoid` and `tf.tanh` is that `tf.tanh` has a range of $[-1.0, 1.0]$. The ability to output negative values may be useful in certain network architectures.

```
# Note, tf.tanh (tf.nn.tanh) is currently limited to float values
features = tf.to_float(tf.range(-1, 3))
sess.run([features, tf.tanh(features)])
```

The output from executing the example code is:

```
[array([-1., 0., 1., 2.], dtype=float32),
 array([-0.76159418, 0., 0.76159418, 0.96402758], dtype=float32)]
```

In this example, all the setup is the same as the `tf.sigmoid` example but the output shows an important difference. In the output of `tf.tanh` the midpoint is 0.0 with negative values. This can cause trouble if the next layer in the network isn't expecting negative input or input of 0.0.

tf.nn.dropout

Set the output to be 0.0 based on a configurable probability. This layer performs well in scenarios where a little randomness helps training. An example scenario is when there are patterns being learned that are too tied to their neighboring features. This layer will add a little noise to the output being learned.

NOTE: This layer should only be used during training because the random noise it adds

will give misleading results while testing.

```
features = tf.constant([-0.1, 0.0, 0.1, 0.2])
# Note, the output should be different on almost ever execution. Your numbers won't match
# this output.
sess.run([features, tf.nn.dropout(features, keep_prob=0.5)])
```

The output from executing the example code is:

```
[array([-0.1, 0., 0.1, 0.2], dtype=float32),
 array([-0., 0., 0.40000001], dtype=float32)]
```

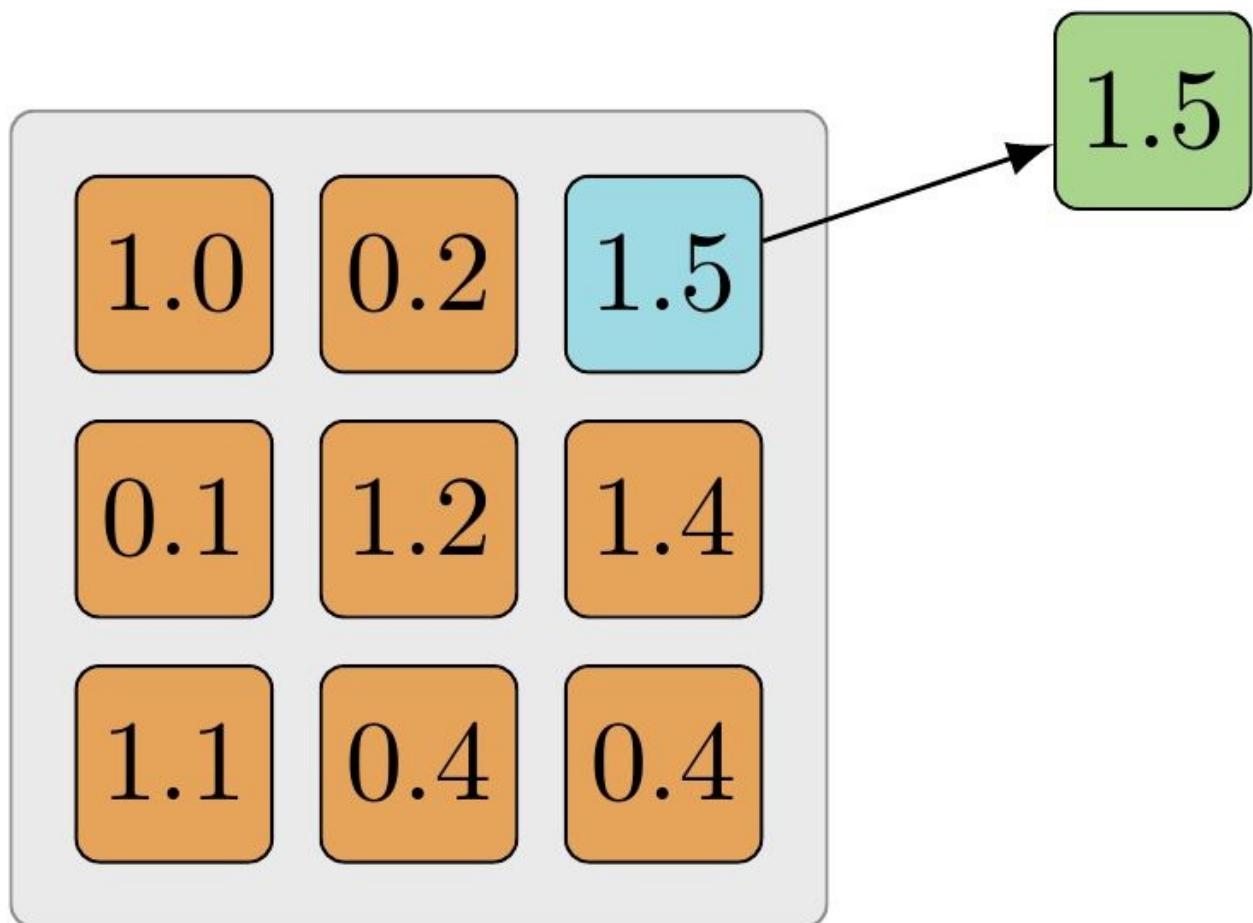
In this example, the output has a 50% probability of being kept. Each execution of this layer will have different output (most likely, it's somewhat random). When an output is dropped, its value is set to 0.0.

Pooling Layers

Pooling layers reduce over-fitting and improving performance by reducing the size of the input. They're used to scale down input while keeping important information for the next layer. It's possible to reduce the size of the input using a `tf.nn.conv2d` alone but these layers execute much faster.

`tf.nn.max_pool`

Strides over a tensor and chooses the maximum value found within a certain kernel size. Useful when the intensity of the input data is relevant to importance in the image.



The same example is modeled using example code below. The goal is to find the largest value within the tensor.

```
# Usually the input would be output from a previous layer and not an image directly.
batch_size=1
input_height = 3
input_width = 3
input_channels = 1

layer_input = tf.constant([
    [
        [[1.0], [0.2], [1.5]],
        [[0.1], [1.2], [1.4]],
        [[1.1], [0.4], [0.4]]
    ]
])

# The strides will look at the entire input by using the image_height and image_width
www.aibbt.com 让未来触手可及
```

```
kernel = [batch_size, input_height, input_width, input_channels]
max_pool = tf.nn.max_pool(layer_input, kernel, [1, 1, 1, 1], "VALID")
sess.run(max_pool)
```

The output from executing the example code is:

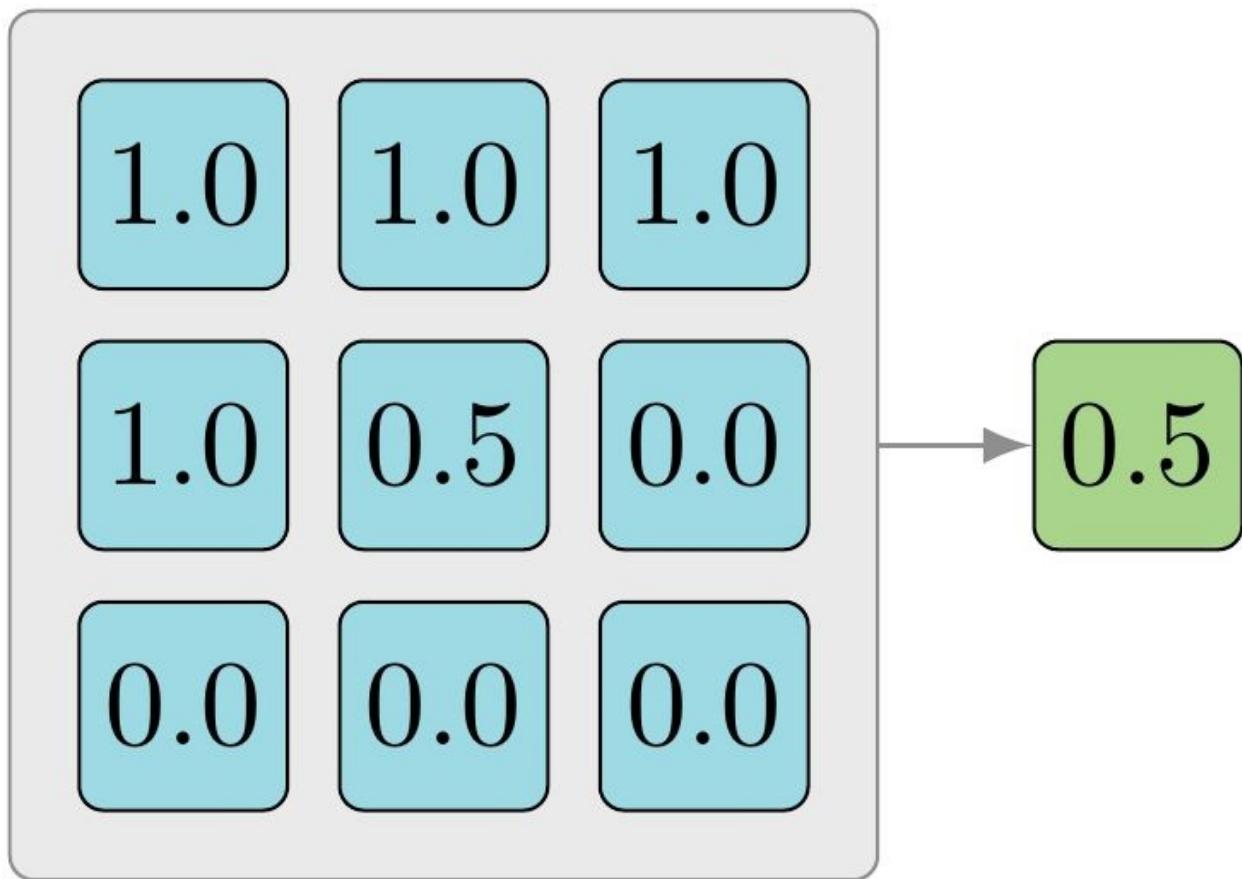
```
array([[[[ 1.5]]]], dtype=float32)
```

The `layer_input` is a tensor with a shape similar to the output of `tf.nn.conv2d` or an activation function. The goal is to keep only one value, the largest value in the tensor. In this case, the largest value of the tensor is 1.5 and is returned in the same format as the input. If the `kernel` were set to be smaller, it would choose the largest value in each kernel size as it strides over the image.

Max-pooling will commonly be done using 2×2 receptive field (kernel with a height of 2 and width of 2) which is often written as a “ 2×2 max-pooling operation”. One reason to use a 2×2 receptive field is that it’s the smallest amount of downsampling which can be done in a single pass. If a 1×1 receptive field were used then the output would be the same as the input.

tf.nn.avg_pool

Strides over a tensor and averages all the values at each depth found within a kernel size. Useful when reducing values where the entire kernel is important, for example, input tensors with a large width and height but small depth.



The same example is modeled using example code below. The goal is to find the

average of all the values within the tensor.

```
batch_size=1
input_height = 3
input_width = 3
input_channels = 1

layer_input = tf.constant([
    [
        [[1.0], [1.0], [1.0]],
        [[1.0], [0.5], [0.0]],
        [[0.0], [0.0], [0.0]]
    ]
])

# The strides will look at the entire input by using the image_height and image_width
kernel = [batch_size, input_height, input_width, input_channels]
max_pool = tf.nn.avg_pool(layer_input, kernel, [1, 1, 1, 1], "VALID")
sess.run(max_pool)
```

The output from executing the example code is:

```
array([[[[ 0.5]]]], dtype=float32)
```

Do a summation of all the values in the tensor, then divide them by the size of the number of scalars in the tensor:

$$\frac{1.0 + 1.0 + 1.0 + 1.0 + 0.5 + 0.0 + 0.0 + 0.0 + 0.0}{9.0}$$

This is exactly what the example code did above, but by reducing the size of the kernel, it's possible to adjust the size of the output.

Normalization

Normalization layers are not unique to CNNs and aren't used as often. When using `tf.nn.relu`, it is useful to consider normalization of the output. Since ReLU is unbounded, it's often useful to utilize some form of normalization to identify high-frequency features.

`tf.nn.local_response_normalization (tf.nn.lrn)`

Local response normalization is a function which shapes the output based on a summation operation best explained in [TensorFlow's documentation](#).

... Within a given vector, each component is divided by the weighted, squared sum of inputs within `depth_radius`.

One goal of normalization is to keep the input in a range of acceptable numbers. For instance, normalizing input in the range of $[0.0, 1.0]$ where the full range of possible values is normalized to be represented by a number greater than or equal to `0.0` and less than or equal to `1.0`. Local response normalization normalizes values while taking into account the significance of each value.

[Cuda-Convet](#) includes further details on why using local response normalization is useful in some CNN architectures. [ImageNet](#) uses this layer to normalize the output from `tf.nn.relu`.

```
# Create a range of 3 floats.
# TensorShape([batch, image_height, image_width, image_channels])
layer_input = tf.constant([
    [[[ 1.]], [[ 2.]], [[ 3.]]]
])

lrn = tf.nn.local_response_normalization(layer_input)
sess.run([layer_input, lrn])
```

The output from executing the example code is:

```
[array([[[[ 1.],
           [[ 2.],
           [[ 3.]]]], dtype=float32), array([[[[ 0.70710677]],
           [[ 0.89442718]],
           [[ 0.94868326]]], dtype=float32)]
```

In this example code, the layer input is in the format `[batch, image_height, image_width, image_channels]`. The normalization reduced the output to be in the range of $[-1.0, 1.0]$. For `tf.nn.relu`, this layer will reduce its unbounded output to be in the same range.

High Level Layers

TensorFlow has introduced high level layers designed to make it easier to create fairly standard layer definitions. These aren't required to use but they help avoid duplicate code while following best practices. While getting started, these layers add a number of non-essential nodes to the graph. It's worth waiting until the basics are comfortable before using these layers.

tf.contrib.layers.convolution2d

The `convolution2d` layer will do the same logic as `tf.nn.conv2d` while including weight initialization, bias initialization, trainable variable output, bias addition and adding an activation function. Many of these steps haven't been covered for CNNs yet but should be familiar. A kernel is a trainable variable (the CNN's goal is to train this variable), weight initialization is used to fill the kernel with values (`tf.truncated_normal`) on its first run. The rest of the parameters are similar to what have been used before except they are reduced to short-hand version. Instead of declaring the full kernel, now it's a simple tuple `(1, 1)` for the kernel's height and width.

```
image_input = tf.constant([
    [
        [[0., 0., 0.], [255., 255., 255.], [254., 0., 0.]],
        [[0., 191., 0.], [3., 108., 233.], [0., 191., 0.]],
        [[254., 0., 0.], [255., 255., 255.], [0., 0., 0.]]
    ]
])

conv2d = tf.contrib.layers.convolution2d(
    image_input,
    num_output_channels=4,
    kernel_size=(1,1),           # It's only the filter height and width.
    activation_fn=tf.nn.relu,
    stride=(1, 1),               # Skips the stride values for image_batch and input_channels.
    trainable=True)

# It's required to initialize the variables used in convolution2d's setup.
sess.run(tf.initialize_all_variables())
sess.run(conv2d)
```

The output from executing the example code is:

```
array([[[[ 0., 0., 0., 0.],
       [ 166.44549561, 0., 0., 0.],
       [ 171.00466919, 0., 0., 0.]],
      [[ 28.54177475, 0., 59.9046936, 0.],
       [ 0., 124.69891357, 0., 0.],
       [ 28.54177475, 0., 59.9046936, 0.]],
      [[ 171.00466919, 0., 0., 0.],
       [ 166.44549561, 0., 0., 0.],
       [ 0., 0., 0., 0.]]]], dtype=float32)
```

This example sets up a full convolution against a batch of a single image. All the parameters are based off of the steps done throughout this chapter. The main difference is that `tf.contrib.layers.convolution2d` does a large amount of setup without having to write it all again. This can be a great time saving layer for advanced users.

NOTE: `tf.to_float` should not be used if the input is an image, instead use `tf.image.convert_image_dtype` which will properly change the range of values used to describe colors. In this example code, float values of 255. were used which aren't what TensorFlow

expects when it sees an image using float values. TensorFlow expects an image with colors described as floats to stay in the range of $[0, 1]$.

tf.contrib.layers.fully_connected

A fully connected layer is one where every input is connected to every output. This is a fairly common layer in many architectures but for CNNs, the last layer is quite often fully connected. The `tf.contrib.layers.fully_connected` layer offers a great short-hand to create this last layer while following best practices.

Typical fully connected layers in TensorFlow are often in the format of `tf.matmul(features, weight) + bias` where `feature`, `weight` and `bias` are all tensors. This short-hand layer will do the same thing while taking care of the intricacies involved in managing the `weight` and `bias` tensors.

```
features = tf.constant([
    [[1.2], [3.4]]
])

fc = tf.contrib.layers.fully_connected(features, num_output_units=2)
# It's required to initialize all the variables first or there'll be an error about precondition fail
sess.run(tf.initialize_all_variables())
sess.run(fc)
```

The output from executing the example code is:

```
array([[[[-0.53210509, 0.74457598],
         [-1.50763106, 2.10963178]]], dtype=float32)
```

This example created a fully connected layer and associated the input tensor with each neuron of the output. There are plenty of other parameters to tweak for different fully connected layers.

Layer Input

Each layer serves a purpose in a CNN architecture. It's important to understand them at a high level (at least) but without practice they're easy to forget. A crucial layer in any neural network is the input layer, where raw input is sent to be trained and tested. For object recognition and classification, the input layer is a `tf.nn.conv2d` layer which accepts images. The next step is to use real images in training instead of example input in the form of `tf.constant` or `tf.range` variables.

Images and TensorFlow

TensorFlow is designed to support working with images as input to neural networks. TensorFlow supports loading common file formats (JPG, PNG), working in different color spaces (RGB, RGBA) and common image manipulation tasks. TensorFlow makes it easier to work with images but it's still a challenge. The largest challenge working with images are the size of the tensor which is eventually loaded. Every image requires a tensor the same size as the image's *height * width * channels*. As a reminder, channels are represented as a rank 1 tensor including a scalar amount of color in each channel.

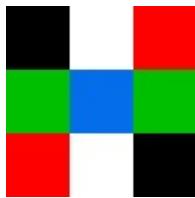
A red RGB pixel in TensorFlow would be represented with the following tensor.

```
red = tf.constant([255, 0, 0])
```

Each scalar can be changed to make the pixel another color or a mix of colors. The rank 1 tensor of a pixel is in the format of [red, green, blue] for an RGB color space. All the pixels in an image are stored in files on a disk which need to be read into memory so TensorFlow may operate on them.

Loading images

TensorFlow is designed to make it easy to load files from disk quickly. Loading images is the same as loading any other large binary file until the contents are decoded. Loading this example 3x3 pixel RGB JPG image is done using a similar process to loading any other type of file.



```
# The match_filenames_once will accept a regex but there is no need for this example.
image_filename = "./images/chapter-05-object-recognition-and-classification/working-with-images/test
filename_queue = tf.train.string_input_producer(
    tf.train.match_filenames_once(image_filename))

image_reader = tf.WholeFileReader()
_, image_file = image_reader.read(filename_queue)
image = tf.image.decode_jpeg(image_file)
```

The image, which is assumed to be located in a relative directory from where this code is ran. An input producer (`tf.train.string_input_producer`) finds the files and adds them to a queue for loading. Loading an image requires loading the entire file into memory (`tf.WholeFileReader`) and once a file has been read (`image_reader.read`) the resulting image is decoded (`tf.image.decode_jpeg`).

Now the image can be inspected, since there is only one file by that name the queue will always return the same image.

```
sess.run(image)
```

The output from executing the example code is:

```
array([[[ 0,  0,  0],
       [255, 255, 255],
       [254,  0,  0]],
      [[ 0, 191,  0],
       [ 3, 108, 233],
       [ 0, 191,  0]],
      [[254,  0,  0],
       [255, 255, 255],
       [ 0,  0,  0]]], dtype=uint8)
```

Inspect the output from loading an image, notice that it's a fairly simple rank 3 tensor. The RGB values are found in 9 rank 1 tensors. The higher rank of the image should be familiar from earlier sections. The format of the image loaded in memory is now

`[batch_size, image_height, image_width, channels]`.

The `batch_size` in this example is 1 because there are no batching operations happening. Batching of input is covered in [the TensorFlow documentation](#) with a great amount of detail. When dealing with images, note the amount of memory required to load the raw images. If the images are too large or too many are loaded in a batch, the system may stop responding.

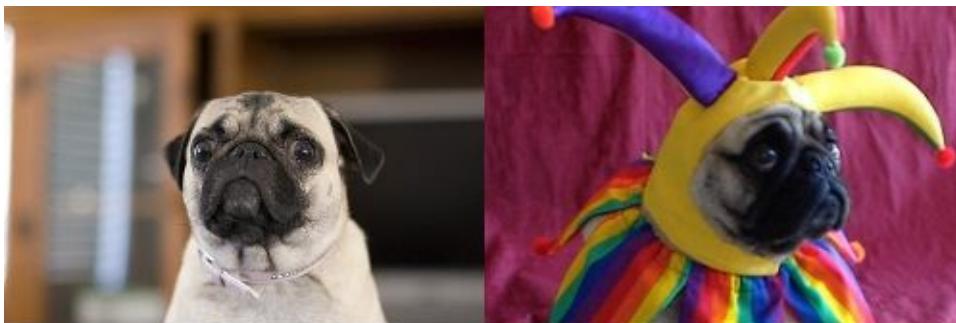
Image Formats

It's important to consider aspects of images and how they affect a model. Consider what would happen if a network is trained with input from a single frame of a [RED Weapon Camera](#), which at the time of writing this, has an effective pixel count of 6144×3160 . That'd be 19,415,040 rank one tensors with 3 dimensions of color information.

Practically speaking, an input of that size will use a huge amount of system memory. Training a CNN takes a large amount of time and loading very large files slow it down more. Even if the increase in time is acceptable, the size a single image would be hard to fit in memory on the majority of system's GPUs.

A large input image is counterproductive to training most CNNs as well. The CNN is attempting to find inherent attributes in an image, which are unique but generalized so that they may be applied to other images with similar results. Using a large input is flooding a network with irrelevant information which will keep from generalizing the model.

In the [Stanford Dogs Dataset](#) there are two extremely different images of the same dog breed which should both match as a Pug. Although cute, these images are filled with useless information which mislead a network during training. For example, the hat worn by the Pug in `n02110958_4030.jpg` isn't a feature a CNN needs to learn in order to match a Pug. Most Pugs prefer pirate hats so the jester hat is training the network to match a hat which most Pugs don't wear.



`n02110958_2410.jpg`

`n02110958_4030.jpg`

Highlighting important information in images is done by storing them in an appropriate file format and manipulating them. Different formats can be used to solve different problems encountered while working with images.

JPEG and PNG

TensorFlow has two image formats used to decode image data, one is `tf.image.decode_jpeg` and the other is `tf.image.decode_png`. These are common file formats in computer vision applications because they're trivial to convert other formats to.

Something important to keep in mind, JPEG images don't store any alpha channel information and PNG images do. This could be important if what you're training on requires alpha information (transparency). An example usage scenario is one where you've manually cut out some pieces of an image, for example, irrelevant jester hats on

dogs. Setting those pieces to black would make them seem of similar importance to other black colored items in the image. Setting the removed hat to have an alpha of 0 would help in distinguishing its removal.

When working with JPEG images, don't manipulate them too much because it'll leave [artifacts](#). Instead, plan to take raw images and export them to JPEG while doing any manipulation needed. Try to manipulate images before loading them whenever possible to save time in training.

PNG images work well if manipulation is required. PNG format is lossless so it'll keep all the information from the original file (unless they've been resized or downsampled). The downside to PNGs is that the files are larger than their JPEG counterpart.

TFRecord

TensorFlow has a built-in file format designed to keep binary data and label (category for training) data in the same file. The format is called TFRecord and the format requires a preprocessing step to [convert images](#) to a TFRecord format before training. The largest benefit is keeping each input image in the same file as the label associated with it.

Technically, TFRecord files are protobuf formatted files. They are great for use as a preprocessed format because they aren't compressed and can be loaded into memory quickly. In this example, an image is written to a new TFRecord formatted file and it's label is stored as well.

```
# Reuse the image from earlier and give it a fake label
image_label = b'\x01' # Assume the label data is in a one-hot representation (00000001)

# Convert the tensor into bytes, notice that this will load the entire image file
image_loaded = sess.run(image)
image_bytes = image_loaded.tobytes()
image_height, image_width, image_channels = image_loaded.shape

# Export TFRecord
writer = tf.python_io.TFRecordWriter("./output/training-image.tfrecord")

# Don't store the width, height or image channels in this Example file to save space but not require
example = tf.train.Example(features=tf.train.Features(feature={
    'label': tf.train.Feature(bytes_list=tf.train.BytesList(value=[image_label])),
    'image': tf.train.Feature(bytes_list=tf.train.BytesList(value=[image_bytes]))
}))

# This will save the example to a text file tfrecord
writer.write(example.SerializeToString())
writer.close()
```

The label is in a format known as one-hot encoding which is a common way to work with label data for categorization of multi-class data. The Stanford Dogs Dataset is being treated as multi-class data because the dogs are being categorized as a single breed and not a mix of breeds. In the real world, a multilabel solution would work well to predict dog breeds because it'd be capable of matching a dog with multiple breeds.

In the example code, the image is loaded into memory and converted into an array of bytes. The bytes are then added to the `tf.train.Example` file which are serialized `serializeToString` before storing to disk. Serialization is a way of converting the in memory

object into a format safe to be transferred to a file. The serialized example is now saved in a format which can be loaded and deserialized back to the example format saved here.

Now that the image is saved as a TFRecord it can be loaded again but this time from the TFRecord file. This would be the loading required in a training step to load the image and label for training. This will save time from loading the input image and its corresponding label separately.

```
# Load TFRecord
tf_record_filename_queue = tf.train.string_input_producer(
    tf.train.match_filenames_once("./output/training-image.tfrecord"))

# Notice the different record reader, this one is designed to work with TFRecord files which may
# have more than one example in them.
tf_record_reader = tf.TFRecordReader()
_, tf_record_serialized = tf_record_reader.read(tf_record_filename_queue)

# The label and image are stored as bytes but could be stored as int64 or float64 values in a
# serialized tf.Example protobuf.
tf_record_features = tf.parse_single_example(
    tf_record_serialized,
    features={
        'label': tf.FixedLenFeature([], tf.string),
        'image': tf.FixedLenFeature([], tf.string),
    })

# Using tf.uint8 because all of the channel information is between 0-255
tf_record_image = tf.decode_raw(
    tf_record_features['image'], tf.uint8)

# Reshape the image to look like the image saved, not required
tf_record_image = tf.reshape(
    tf_record_image,
    [image_height, image_width, image_channels])
# Use real values for the height, width and channels of the image because it's required
# to reshape the input.

tf_record_label = tf.cast(tf_record_features['label'], tf.string)
```

At first, the file is loaded in the same way as any other file. The main difference is that the file is then read using a `TFRecordReader`. Instead of decoding the image, the TFRecord is parsed `tf.parse_single_example` and then the image is read as raw bytes (`tf.decode_raw`).

After the file is loaded, it is reshaped (`tf.reshape`) in order to keep it in the same layout as `tf.nn.conv2d` expects it `[image_height, image_width, image_channels]`. It'd be safe to expand the dimensions (`tf.expand`) in order to add in the `batch_size` dimension to the `input_batch`.

In this case a single image is in the TFRecord but these record files support multiple examples being written to them. It'd be safe to have a single TFRecord file which stores an entire training set but splitting up the files doesn't hurt.

The following code is useful to check that the image saved to disk is the same as the image which was loaded from TensorFlow.

```
sess.run(tf.equal(image, tf_record_image))
```

The output from executing the example code is:

```
array([[[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]],
      [[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]]],
```

```
[[ True, True, True],  
 [ True, True, True],  
 [ True, True, True]]], dtype=bool)
```

All of the attributes of the original image and the image loaded from the TFRecord file are the same. To be sure, load the label from the TFRecord file and check that it is the same as the one saved earlier.

```
# Check that the label is still 0b00000001.  
sess.run(tf_record_label)
```

The output from executing the example code is:

```
b'\x01'
```

Creating a file that stores both the raw image data and the expected output label will save complexities during training. It's not required to use TFRecord files but it's highly recommend when working with images. If it doesn't work well for a workflow, it's still recommended to preprocess images and save them before training. Manipulating an image each time it's loaded is not recommended.

Image Manipulation

CNNs work well when they're given a large amount of diverse quality training data. Images capture complex scenes in a way which visually communicates an intended subject. In the Stanford Dog's Dataset, it's important that the images visually highlight the importance of dogs in the picture. A picture with a dog clearly visible in the center is considered more valuable than one with a dog in the background.

Not all datasets have the most valuable images. The following are two images from the [Stanford Dogs Dataset](#), which are supposed to highlight dog breeds. The image on the left `n02113978_3480.jpg` highlights important attributes of a typical Mexican Hairless Dog, while the image on the right `n02113978_1030.jpg` highlights the look of inebriated party goers scaring a Mexican Hairless Dog. The image on the right `n02113978_1030.jpg` is filled with irrelevant information which may train a CNN to categorize party goer faces instead of Mexican Hairless Dog breeds. Images like this may still include an image of a dog and could be manipulated to highlight the dog instead of people.



Image manipulation is best done as a preprocessing step in most scenarios. An image can be cropped, resized and the color levels adjusted. On the other hand, there is an important use case for manipulating an image while training. After an image is loaded, it can be flipped or distorted to diversify the input training information used with the network. This step adds further processing time but helps with overfitting.

TensorFlow is not designed as an image manipulation framework. There are libraries available in Python which support more image manipulation than TensorFlow ([PIL](#) and [OpenCV](#)). For TensorFlow, we'll summarize a few useful image manipulation features available which are useful in training CNNs.

Cropping

Cropping an image will remove certain regions of the image without keeping any information. Cropping is similar to `tf.slice` where a section of a tensor is cut out from the full tensor. Cropping an input image for a CNN can be useful if there is extra input along a dimension which isn't required. For example, cropping dog pictures where the dog is in the center of the images to reduce the size of the input.

```
sess.run(tf.image.central_crop(image, 0.1))
```

The output from executing the example code is:

```
array([[[ 3, 108, 233]]], dtype=uint8)
```

The example code uses `tf.image.central_crop` to crop out 10% of the image and return it. This method always returns based on the center of the image being used.

Cropping is usually done in preprocessing but it can be useful when training if the background is useful. When the background is useful then cropping can be done while randomizing the center offset of where the crop begins.

```
# This crop method only works on real value input.
real_image = sess.run(image)

bounding_crop = tf.image.crop_to_bounding_box(
    real_image, offset_height=0, offset_width=0, target_height=2, target_width=1)

sess.run(bounding_crop)
```

The output from executing the example code is:

```
array([[[ 0, 0, 0]],
   [[ 0, 191, 0]]], dtype=uint8)
```

The example code uses `tf.image.crop_to_bounding_box` in order to crop the image starting at the upper left pixel located at $(0, 0)$. Currently, the function only works with a tensor which has a defined shape so an input image needs to be executed on the graph first.

Padding

Pad an image with zeros in order to make it the same size as an expected image. This can be accomplished using `tf.pad` but TensorFlow has another function useful for resizing images which are too large or too small. The method will pad an image which is too small including zeros along the edges of the image. Often, this method is used to resize small images because any other method of resizing will distort the image.

```
# This padding method only works on real value input.
real_image = sess.run(image)

pad = tf.image.pad_to_bounding_box(
    real_image, offset_height=0, offset_width=0, target_height=4, target_width=4)

sess.run(pad)
```

The output from executing the example code is:

```
array([[[ 0, 0, 0],
   [255, 255, 255],
   [254, 0, 0],
   [ 0, 0, 0]],
  [[ 0, 191, 0],
   [ 3, 108, 233],
   [ 0, 191, 0],
   [ 0, 0, 0]],
  [[254, 0, 0],
   [255, 255, 255],
   [ 0, 0, 0],
   [ 0, 0, 0]],
  [[ 0, 0, 0],
   [ 0, 0, 0],
   [ 0, 0, 0],
   [ 0, 0, 0]]], dtype=uint8)
```

This example code increases the images height by one pixel and its width by a pixel as well. The new pixels are all set to 0. Padding in this manner is useful for scaling up an image which is too small. This can happen if there are images in the training set with a

mix of aspect ratios. TensorFlow has a useful shortcut for resizing images which don't match the same aspect ratio using a combination of `pad` and `crop`.

```
# This padding method only works on real value input.
real_image = sess.run(image)

crop_or_pad = tf.image.resize_image_with_crop_or_pad(
    real_image, target_height=2, target_width=5)

sess.run(crop_or_pad)
```

The output from executing the example code is:

```
array([[[ 0,  0,  0],
       [ 0,  0,  0],
       [255, 255, 255],
       [254,  0,  0],
       [ 0,  0,  0]],
      [[ 0,  0,  0],
       [ 0, 191,  0],
       [ 3, 108, 233],
       [ 0, 191,  0],
       [ 0,  0,  0]]], dtype=uint8)
```

The `real_image` has been reduced in height to be 2 pixels tall and the width has been increased by padding the image with zeros. This function works based on the center of the image input.

Flipping

Flipping an image is exactly what it sounds like. Each pixel's location is reversed horizontally or vertically. Technically speaking, flopping is the term used when flipping an image vertically. Terms aside, flipping images is useful with TensorFlow to give different perspectives of the same image for training. For example, a picture of an Australian Shepherd with crooked left ear could be flipped in order to allow matching of crooked right ears.

TensorFlow has functions to flip images vertically, horizontally and choose randomly. The ability to randomly flip an image is a useful method to keep from overfitting a model to flipped versions of images.

```
top_left_pixels = tf.slice(image, [0, 0, 0], [2, 2, 3])
flip_horizontal = tf.image.flip_left_right(top_left_pixels)
flip_vertical = tf.image.flip_up_down(flip_horizontal)

sess.run([top_left_pixels, flip_vertical])
```

The output from executing the example code is:

```
[array([[[ 0,  0,  0],
       [255, 255, 255]],
      [[ 0, 191,  0],
       [ 3, 108, 233]]], dtype=uint8), array([[[ 3, 108, 233],
       [ 0, 191,  0]],
      [[255, 255, 255],
       [ 0,  0,  0]]], dtype=uint8)]
```

This example code flips a subset of the image horizontally and then vertically. The subset is used with `tf.slice` because the original image flipped returns the same images (for this example only). The subset of pixels illustrates the change which occurs when an image is flipped. `tf.image.flip_left_right` and `tf.image.flip_up_down` both operate on tensors

which are not limited to images. These will flip an image a single time, randomly flipping an image is done using a separate set of functions.

```
top_left_pixels = tf.slice(image, [0, 0, 0], [2, 2, 3])
random_flip_horizontal = tf.image.random_flip_left_right(top_left_pixels)
random_flip_vertical = tf.image.random_flip_up_down(random_flip_horizontal)
sess.run(random_flip_vertical)
```

The output from executing the example code is:

```
array([[[ 3, 108, 233],
       [ 0, 191, 0]],
      [[255, 255, 255],
       [ 0, 0, 0]]], dtype=uint8)
```

This example does the same logic as the example before except that the output is random. Every time this runs, a different output is expected. There is a parameter named `seed` which may be used to control how random the flipping occurs.

Saturation and Balance

Images which are found on the internet are often edited in advance. For instance, many of the images found in the Stanford Dogs dataset have too much saturation (lots of color). When an edited image is used for training, it may mislead a CNN model into finding patterns which are related to the edited image and not the content in the image.

TensorFlow has useful functions which help in training on images by changing the saturation, hue, contrast and brightness. The functions allow for simple manipulation of these image attributes as well as randomly altering these attributes. The random altering is useful in training in for the same reason randomly flipping an image is useful. The random attribute changes help a CNN be able to accurately match a feature in images which have been edited or were taken under different lighting.

```
example_red_pixel = tf.constant([254., 2., 15.])
adjust_brightness = tf.image.adjust_brightness(example_red_pixel, 0.2)
sess.run(adjust_brightness)
```

The output from executing the example code is:

```
array([ 254.19999695, 2.20000005, 15.19999981], dtype=float32)
```

This example brightens a single pixel, which is primarily red, with a delta of `0.2`. Unfortunately, in the current version of TensorFlow 0.9, this method doesn't work well with a `tf.uint8` input. It's best to avoid using this when possible and preprocess brightness changes.

```
adjust_contrast = tf.image.adjust_contrast(image, -0.5)
sess.run(tf.slice(adjust_contrast, [1, 0, 0], [1, 3, 3]))
```

The output from executing the example code is:

```
array([[[170, 71, 124],
       [168, 112, 7],
       [170, 71, 124]]], dtype=uint8)
```

The example code changes the contrast by `-0.5` which makes the new version of the

image fairly unrecognizable. Adjusting contrast is best done in small increments to keep from blowing out an image. Blowing out an image means the same thing as saturating a neuron, it reached its maximum value and can't be recovered. With contrast changes, an image can become completely white and completely black from the same adjustment.

The `tf.slice` operation is for brevity, highlighting one of the pixels which has changed. It is not required when running this operation.

```
adjust_hue = tf.image.adjust_hue(image, 0.7)
sess.run(tf.slice(adjust_hue, [1, 0, 0], [1, 3, 3]))
```

The output from executing the example code is:

```
array([[[191, 38, 0],
       [62, 233, 3],
       [191, 38, 0]]], dtype=uint8)
```

The example code adjusts the hue found in the image to make it more colorful. The adjustment accepts a `delta` parameter which controls the amount of hue to adjust in the image.

```
adjust_saturation = tf.image.adjust_saturation(image, 0.4)
sess.run(tf.slice(adjust_saturation, [1, 0, 0], [1, 3, 3]))
```

The output from executing the example code is:

```
array([[[114, 191, 114],
       [141, 183, 233],
       [114, 191, 114]]], dtype=uint8)
```

The code is similar to adjusting the contrast. It is common to oversaturate an image in order to identify edges because the increased saturation highlights changes in colors.

Colors

CNNs are commonly trained using images with a single color. When an image has a single color it is said to use a grayscale colorspace meaning it uses a single channel of colors. For most computer vision related tasks, using grayscale is reasonable because the shape of an image can be seen without all the colors. The reduction in colors equates to a quicker to train image. Instead of a 3 component rank 1 tensor to describe each color found with RGB, a grayscale image requires a single component rank 1 tensor to describe the amount of gray found in the image.

Although grayscale has benefits, it's important to consider applications which require a distinction based on color. Color in images is challenging to work with in most computer vision because it isn't easy to mathematically define the similarity of two RGB colors. In order to use colors in CNN training, it's useful to convert the colorspace the image is natively in.

Grayscale

Grayscale has a single component to it and has the same range of color as RGB $[0, 255]$.

```
gray = tf.image.rgb_to_grayscale(image)
sess.run(tf.slice(gray, [0, 0, 0], [1, 3, 1]))
```

The output from executing the example code is:

```
array([[[ 0],
       [255],
       [ 76]]], dtype=uint8)
```

This example converted the RGB image into grayscale. The `tf.slice` operation took the top row of pixels out to investigate how their color has changed. The grayscale conversion is done by averaging all the color values for a pixel and setting the amount of grayscale to be the average.

HSV

Hue, saturation and value are what make up HSV colorspace. This space is represented with a 3 component rank 1 tensor similar to RGB. HSV is not similar to RGB in what it measures, it's measuring attributes of an image which are closer to human perception of color than RGB. It is sometimes called HSB, where the B stands for brightness.

```
hsv = tf.image.rgb_to_hsv(tf.image.convert_image_dtype(image, tf.float32))
sess.run(tf.slice(hsv, [0, 0, 0], [3, 3, 3]))
```

The output from executing the example code is:

```
array([[[ 0.,  0.,  0.],
       [ 0.,  0.,  1.],
       [ 0.,  1.,  0.99607849]],
      [[ 0.33333334,  1.,  0.74901962],
       [ 0.59057975,  0.98712444,  0.91372555],
       [ 0.33333334,  1.,  0.74901962]],
      [[ 0.,  1.,  0.99607849],
```

```
[ 0., 0., 1.],  
[ 0., 0., 0.]], dtype=float32)
```

RGB

RGB is the colorspace which has been used in all the example code so far. It's broken up into a 3 component rank 1 tensor which includes the amount of red [0, 255], green [0, 255] and blue [0, 255]. Most images are already in RGB but TensorFlow has builtin functions in case the images are in another colorspace.

```
rgb_hsv = tf.image.hsv_to_rgb(hsv)  
rgb_grayscale = tf.image.grayscale_to_rgb(gray)
```

The example code is straightforward except that the conversion from grayscale to RGB doesn't make much sense. RGB expects three colors while grayscale only has one. When the conversion occurs, the RGB values are filled with the same value which is found in the grayscale pixel.

Lab

Lab is not a colorspace which TensorFlow has native support for. It's a useful colorspace because it can map to a larger number of perceivable colors than RGB. Although TensorFlow doesn't support this natively, it is a colorspace, which is often used in professional settings. Another Python library [python-colormath](#) has support for Lab conversion as well as other colorspaces not described here.

The largest benefit using a Lab colorspace is it maps closer to humans perception of the difference in colors than RGB or HSV. The euclidean distance between two colors in a Lab colorspace are somewhat representative of how different the colors look to a human.

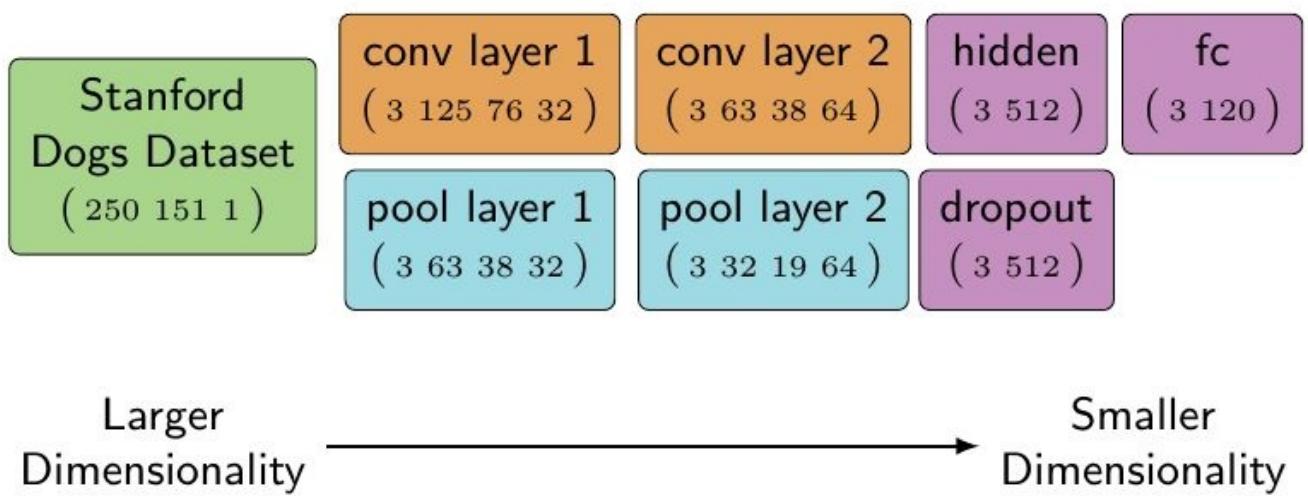
Casting Images

In these examples, `tf.to_float` is often used in order to illustrate changing an image's type to another format. For example, this works OK but TensorFlow has a built in function to properly scale values as they change types. `tf.image.convert_image_dtype(image, dtype, saturate=False)` is a useful shortcut to change the type of an image from `tf.uint8` to `tf.float`.

CNN Implementation

Object recognition and categorization using TensorFlow required a basic understanding of convolutions (for CNNs), common layers (non-linearity, pooling, fc), image loading, image manipulation and colorspace. With these areas covered, it's possible to build a CNN model for image recognition and classification using TensorFlow. In this case, the model is a dataset provided by Stanford which includes pictures of dogs and their corresponding breed. The network needs to train on these pictures then be judged on how well it can guess a dog's breed based on a picture.

The network architecture follows a simplified version of [Alex Krizhevsky's AlexNet](#) without all of AlexNet's layers. This architecture was described earlier in the chapter as the network which won ILSVRC'12 top challenge. The network uses layers and techniques familiar to this chapter which are similar to the [TensorFlow provided](#) tutorial on CNNs.



The network described in this section including the output TensorShape after each layer. The layers are read from left to right and top to bottom where related layers are grouped together. As the input progresses further into the network, its height and width are reduced while its depth is increased. The increase in depth reduces the computation required to use the network.

Stanford Dogs Dataset

The dataset used for training this model can be found on Stanford's computer vision site <http://vision.stanford.edu/aditya86/ImageNetDogs/>. Training the model requires downloading relevant data. After downloading the Zip archive of all the images, extract the archive into a new directory called `imagenet-dogs` in the same directory as the code building the model.

The Zip archive provided by Stanford includes pictures of dogs organized into 120 different breeds. The goal of this model is to train on 80% of the dog breed images and then test using the remaining 20%. If this were a production model, part of the raw data would be reserved for cross-validation of the results. Cross-validation is a useful step to validate the accuracy of a model but this model is designed to illustrate the process and not for competition.

The organization of the archive follows ImageNet's practices. Each dog breed is a directory name similar to `n02085620-chihuahua` where the second half of the directory name is the dog's breed in English (`chihuahua`). Within each directory there is a variable amount of images related to that breed. Each image is in JPEG format (RGB) and of varying sizes. The different sized images is a challenge because TensorFlow is expecting tensors of the same dimensionality.

Convert Images to TFRecords

The raw images organized in a directory doesn't work well for training because the images are not of the same size and their dog breed isn't included in the file. Converting the images into TFRecord files in advance of training will help keep training fast and simplify matching the label of the image. Another benefit is that the training and testing related images can be separated in advance. Separated training and testing datasets allows continual testing of a model while training is occurring using checkpoint files.

Converting the images will require changing their colorspace into grayscale, resizing the images to be of uniform size and attaching the label to each image. This conversion should only happen once before training commences and likely will take a long time.

```
import glob

image_filenames = glob.glob("./imagenet-dogs/n02*/*.jpg")
image_filenames[0:2]
```

The output from executing the example code is:

```
['./imagenet-dogs/n02085620-Chihuahua/n02085620_10074.jpg',
 './imagenet-dogs/n02085620-Chihuahua/n02085620_10131.jpg']
```

An example of how the archive is organized. The `glob` module allows directory listing which shows the structure of the files which exist in the dataset. The eight digit number is tied to the [WordNet ID](#) of each category used in ImageNet. ImageNet has a browser for image details which accepts the WordNet ID, for example the Chihuahua example can be accessed via <http://www.image-net.org/synset?wnid=n02085620>.

```
from itertools import groupby
from collections import defaultdict

training_dataset = defaultdict(list)
testing_dataset = defaultdict(list)

# Split up the filename into its breed and corresponding filename. The breed is found by taking the
image_filename_with_breed = map(lambda filename: (filename.split("/")[2], filename), image_filenames)

# Group each image by the breed which is the 0th element in the tuple returned above
for dog_breed, breed_images in groupby(image_filename_with_breed, lambda x: x[0]):
    # Enumerate each breed's image and send ~20% of the images to a testing set
    for i, breed_image in enumerate(breed_images):
        if i % 5 == 0:
            testing_dataset[dog_breed].append(breed_image[1])
        else:
            training_dataset[dog_breed].append(breed_image[1])

# Check that each breed includes at least 18% of the images for testing
breed_training_count = len(training_dataset[dog_breed])
breed_testing_count = len(testing_dataset[dog_breed])

assert round(breed_testing_count / (breed_training_count + breed_testing_count), 2) > 0.18, "Not
```

This example code organized the directory and images ('./imagenet-dogs/n02085620-Chihuahua/n02085620_10131.jpg') into two dictionaries related to each breed including all the images for that breed. Now each dictionary would include Chihuahua images in the following format:

```
training_dataset["n02085620-Chihuahua"] = ["n02085620_10131.jpg", ...]
```

Organizing the breeds into these dictionaries simplifies the process of selecting each

type of image and categorizing it. During preprocessing, all the image breeds can be iterated over and their images opened based on the filenames in the list.

```
def write_records_file(dataset, record_location):
    """
    Fill a TFRecords file with the images found in `dataset` and include their category.

    Parameters
    -----
    dataset : dict(list)
        Dictionary with each key being a label for the list of image filenames of its value.
    record_location : str
        Location to store the TFRecord output.
    """
    writer = None

    # Enumerating the dataset because the current index is used to breakup the files if they get over
    # images to avoid a slowdown in writing.
    current_index = 0
    for breed, images_filenames in dataset.items():
        for image_filename in images_filenames:
            if current_index % 100 == 0:
                if writer:
                    writer.close()

            record_filename = "{record_location}-{current_index}.tfrecords".format(
                record_location=record_location,
                current_index=current_index)

            writer = tf.python_io.TFRecordWriter(record_filename)
            current_index += 1

            image_file = tf.read_file(image_filename)

            # In ImageNet dogs, there are a few images which TensorFlow doesn't recognize as JPEGs.
            # try/catch will ignore those images.
            try:
                image = tf.image.decode_jpeg(image_file)
            except:
                print(image_filename)
                continue

            # Converting to grayscale saves processing and memory but isn't required.
            grayscale_image = tf.image.rgb_to_grayscale(image)
            resized_image = tf.image.resize_images(grayscale_image, 250, 151)

            # tf.cast is used here because the resized images are floats but haven't been converted
            # to image floats where an RGB value is between [0,1].
            image_bytes = sess.run(tf.cast(resized_image, tf.uint8)).tobytes()

            # Instead of using the label as a string, it'd be more efficient to turn it into either
            # integer index or a one-hot encoded rank one tensor.
            # https://en.wikipedia.org/wiki/One-hot
            image_label = breed.encode("utf-8")

            example = tf.train.Example(features=tf.train.Features(feature={
                'label': tf.train.Feature(bytes_list=tf.train.BytesList(value=[image_label])),
                'image': tf.train.Feature(bytes_list=tf.train.BytesList(value=[image_bytes]))
            }))

            writer.write(example.SerializeToString())
    writer.close()

write_records_file(testing_dataset, "./output/testing-images/testing-image")
write_records_file(training_dataset, "./output/training-images/training-image")
```

The example code is opening each image, converting it to grayscale, resizing it and then adding it to a TFRecord file. The logic isn't different from earlier examples except that the operation `tf.image.resize_images` is used. The resizing operation will scale every image to be the same size even if it distorts the image. For example, if an image in portrait orientation and an image in landscape orientation were both resized with this code then the output of

the landscape image would become distorted. These distortions are caused because `tf.image.resize_images` doesn't take into account aspect ratio (the ratio of height to width) of an image. To properly resize a set of images, cropping or padding is a preferred method because it ignores the aspect ratio stopping distortions.

Load Images

Once the testing and training dataset have been transformed to TFRecord format, they can be read as TFRecords instead of as JPEG images. The goal is to load the images a few at a time with their corresponding labels.

```
filename_queue = tf.train.string_input_producer(
    tf.train.match_filenames_once("./output/training-images/*.tfrecords"))
reader = tf.TFRecordReader()
_, serialized = reader.read(filename_queue)

features = tf.parse_single_example(
    serialized,
    features={
        'label': tf.FixedLenFeature([], tf.string),
        'image': tf.FixedLenFeature([], tf.string),
    })
record_image = tf.decode_raw(features['image'], tf.uint8)

# Changing the image into this shape helps train and visualize the output by converting it to
# be organized like an image.
image = tf.reshape(record_image, [250, 151, 1])

label = tf.cast(features['label'], tf.string)

min_after_dequeue = 10
batch_size = 3
capacity = min_after_dequeue + 3 * batch_size
image_batch, label_batch = tf.train.shuffle_batch(
    [image, label], batch_size=batch_size, capacity=capacity, min_after_dequeue=min_after_dequeue)
```

This example code loads training images by matching all the TFRecord files found in the training directory. Each TFRecord includes multiple images but the `tf.parse_single_example` will take a single example out of the file. The batching operation discussed earlier is used to train multiple images simultaneously. Batching multiple images is useful because these operations are designed to work with multiple images the same as with a single image. The primary requirement is that the system have enough memory to work with them all.

With the images available in memory, the next step is to create the model used for training and testing.

Model

The model used is similar to the [mnist convolution example](#) which is often used in tutorials describing convolutional neural networks in TensorFlow. The architecture of this model is simple yet it performs well for illustrating different techniques used in image classification and recognition. An advanced model may borrow more from [Alex Krizhevsky's AlexNet](#) design that includes more convolution layers.

```
# Converting the images to a float of [0,1) to match the expected input to convolution2d
float_image_batch = tf.image.convert_image_dtype(image_batch, tf.float32)

conv2d_layer_one = tf.contrib.layers.convolution2d(
    float_image_batch,
    num_output_channels=32,           # The number of filters to generate
    kernel_size=(5,5),                # It's only the filter height and width.
    activation_fn=tf.nn.relu,
    weight_init=tf.random_normal,
    stride=(2, 2),
    trainable=True)
pool_layer_one = tf.nn.max_pool(conv2d_layer_one,
    ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1],
    padding='SAME')

# Note, the first and last dimension of the convolution output hasn't changed but the
# middle two dimensions have.
conv2d_layer_one.get_shape(), pool_layer_one.get_shape()
```

The output from executing the example code is:

```
(TensorShape([Dimension(3), Dimension(125), Dimension(76), Dimension(32)]),
 TensorShape([Dimension(3), Dimension(63), Dimension(38), Dimension(32)]))
```

The first layer in the model is created using the shortcut `tf.contrib.layers.convolution2d`. It's important to note that the `weight_init` is set to be a random normal, meaning that the first set of filters are filled with random numbers following a normal distribution (this parameter is renamed in TensorFlow 0.9 to be `weights_initializer`). The filters are set as `trainable` so that as the network is fed information, these weights are adjusted to improve the accuracy of the model.

After a convolution is applied to the images, the output is downsized using a `max_pool` operation. After the operation, the output shape of the convolution is reduced in half due to the `ksize` used in the pooling and the `strides`. The reduction didn't change the number of filters (output channels) or the size of the image batch. The components that were reduced dealt with the height and width of the image (filter).

```
conv2d_layer_two = tf.contrib.layers.convolution2d(
    pool_layer_one,
    num_output_channels=64,           # More output channels means an increase in the number of filters
    kernel_size=(5,5),
    activation_fn=tf.nn.relu,
    weight_init=tf.random_normal,
    stride=(1, 1),
    trainable=True)

pool_layer_two = tf.nn.max_pool(conv2d_layer_two,
    ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1],
    padding='SAME')

conv2d_layer_two.get_shape(), pool_layer_two.get_shape()
```

The output from executing the example code is:

```
(TensorShape([Dimension(3), Dimension(63), Dimension(38), Dimension(64)]),
 TensorShape([Dimension(3), Dimension(32), Dimension(19), Dimension(64)]))
```

The second layer changes little from the first except the depth of the filters. The number of filters is now doubled while again reducing the size of the height and width of the image. The multiple convolution and pool layers are continuing to reduce the height and width of the input while adding further depth.

At this point, further convolution and pool steps could be taken. In many architectures there are over 5 different convolution and pooling layers. The most advanced architectures take longer to train and debug but they can match more sophisticated patterns. In this example, the two convolution and pooling layers are enough to illustrate the mechanics at work.

The tensor being operated on is still fairly complex tensor, the next step is to fully connect every point in each image with an output neuron. Since this example is using softmax later, the fully connected layer needs to be changed into a rank two tensor. The tensor's first dimension will be used to separate each image while the second dimension is a rank one tensor of each input tensor.

```
flattened_layer_two = tf.reshape(
    pool_layer_two,
    [
        batch_size, # Each image in the image_batch
        -1          # Every other dimension of the input
    ])
flattened_layer_two.get_shape()
```

The output from executing the example code is:

```
TensorShape([Dimension(3), Dimension(38912)])
```

`tf.reshape` has a special value that can be used to signify, use all the dimensions remaining. In this example code, the `-1` is used to reshape the last pooling layer into a giant rank one tensor. With the pooling layer flattened out, it can be combined with two fully connected layers which associate the current network state to the breed of dog predicted.

```
# The weight_init parameter can also accept a callable, a lambda is used here returning a truncated
# with a stddev specified.
hidden_layer_three = tf.contrib.layers.fully_connected(
    flattened_layer_two,
    512,
    weight_init=lambda i, dtype: tf.truncated_normal([38912, 512], stddev=0.1),
    activation_fn=tf.nn.relu
)
# Dropout some of the neurons, reducing their importance in the model
hidden_layer_three = tf.nn.dropout(hidden_layer_three, 0.1)

# The output of this are all the connections between the previous layers and the 120 different dog b
# available to train on.
final_fully_connected = tf.contrib.layers.fully_connected(
    hidden_layer_three,
    120, # Number of dog breeds in the ImageNet Dogs dataset
    weight_init=lambda i, dtype: tf.truncated_normal([512, 120], stddev=0.1)
)
```

This example code creates the final fully connected layer of the network where every pixel is associated with every breed of dog. Every step of this network has been reducing the size of the input images by converting them into filters which are then matched with a

breed of dog (label). This technique has reduced the processing power required to train or test a network while generalizing the output.

Training

Once a model is ready to be trained, the last steps follow the same process discussed in earlier chapters of this book. The model's loss is computed based on how accurately it guessed the correct labels in the training data which feeds into a training optimizer which updates the weights of each layer. This process continues one iteration at a time while attempting to increase the accuracy of each step.

An important note related to this model, during training most classification functions (`tf.nn.softmax`) require numerical labels. This was highlighted in the section describing loading the images from TFRecords. At this point, each label is a string similar to `n02085620-Chihuahua`. Instead of using `tf.nn.softmax` on this string, the label needs to be converted to be a unique number for each label. Converting these labels into an integer representation should be done in preprocessing.

For this dataset, each label will be converted into an integer which represents the index of each name in a list including all the dog breeds. There are many ways to accomplish this task, for this example a new TensorFlow utility operation will be used (`tf.map_fn`).

```
import glob

# Find every directory name in the imagenet-dogs directory (n02085620-Chihuahua, ...)
labels = list(map(lambda c: c.split("/")[-1], glob.glob("./imagenet-dogs/*")))

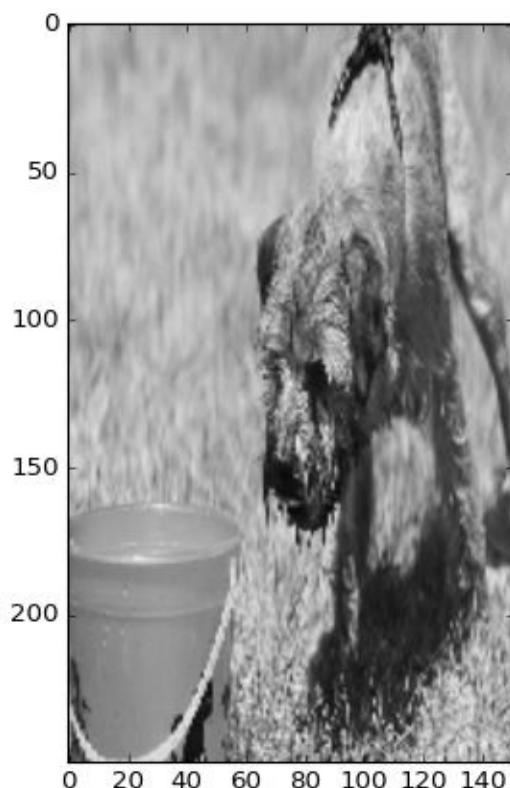
# Match every label from label_batch and return the index where they exist in the list of classes
train_labels = tf.map_fn(lambda l: tf.where(tf.equal(labels, l))[0,0:1][0], label_batch, dtype=tf.in
```

This example code uses two different forms of a `map` operation. The first form of `map` is used to create a list including only the dog breed name based on a list of directories. The second form of `map` is `tf.map_fn` which is a TensorFlow operation that will map a function over a tensor on the graph. The `tf.map_fn` is used to generate a rank one tensor including only the integer indexes where each label is located in the list of all the class labels. These unique integers can now be used with `tf.nn.softmax` to classify output predictions.

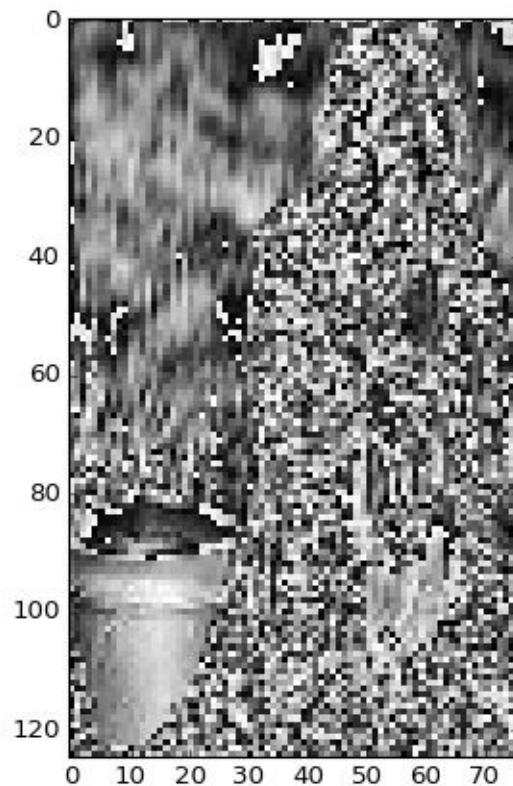
Debug the Filters with Tensorboard

CNNs have multiple moving parts which can cause issues during training resulting in poor accuracy. Debugging problems in a CNN often start with investigating how the filters (kernels) are changing every iteration. Each weight used in a filter is constantly changing as the network attempts to learn the most accurate set of weights to use based on the train method.

In a well designed CNN, when the first convolution layer is started, the initialized input weights are set to be random (in this case using `weight_init=tf.random_normal`). These weights activate over an image and the output of the activation (feature map) is random as well. Visualizing the feature map as if it were an image, the output looks like the original image with static applied. The static is caused by all the weights activating at random. Over many iterations, each filter becomes more uniform as the weights are adjusted to fit the training feedback. As the network converges, the filters resemble distinct small patterns which can be found in the image. Here is an original grayscale training image before it is passed through the first convolution layer:



And, here is a single feature map from the first convolution layer highlighting randomness in the output:



Debugging a CNN requires a familiarity working with these filters. Currently there isn't any built in support in tensorboard to display filters or feature maps. A simple view of the filters can be done using a `tf.image_summary` operation on the filters being trained and the feature maps generated. Adding an image summary output to a graph gives a good overview of the filters being used and the feature map generated by applying them to the input images.

The Jupyter notebook extension worth mentioning is [TensorDebugger](#), which is in an early state of development. The extension has a mode capable of viewing changes in filters as an animated Gif over iterations.

Conclusion

Convolutional Neural Networks are a useful network architecture that are implemented with a minimal amount of code in TensorFlow. While they're designed with images in mind, a CNN is not limited to image input. Convolutions are used in multiple industries from music to medical and a CNN can be applied in a similar manner. Currently, TensorFlow is designed for two dimensional convolutions but it's still possible to work with higher dimensionality input using TensorFlow.

While a CNN could theoretically work with natural language data (text), it isn't designed for this type of input. Text input is often stored in a `SparseTensor` where the majority of the input is 0. CNNs are designed to work with dense input where each value is important and the majority of the input is not 0. Working with text data is a challenge which is addressed in the next chapter on "Recurrent Neural Networks and Natural Language Processing".

Chapter 6. Recurrent Neural Networks and Natural Language Processing

In the previous chapter, we learned to classify static images. This is a huge application of machine learning, but there is more. In this chapter, we will take a look at sequential models. Those models are model powerful in a way, allowing us to classify or label sequential inputs, generate sequences of text or translate one sequence into another.

What we learn here is not distinct from static classification and regression. Recurrent neural networks provide building blocks that fit well into the toolkit of fully connected and convolutional layers. But let's start with the basics.

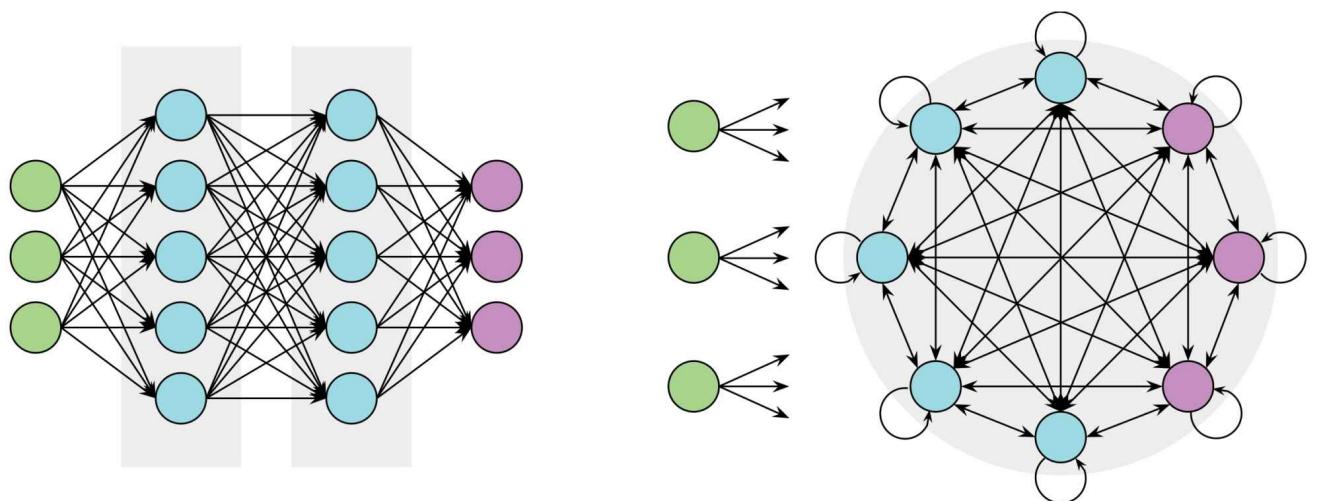
Introduction to Recurrent Networks

Many real-world problems are sequential in nature. This includes almost all problems in natural language processing (NLP). Paragraphs are sequences of sentences, sentences are sequences of words, and words are sequences of characters. Closely related, audio and video clips are sequences of frames changing over time. And even stock market prices only make sense when analyzed over time (if at all).

In all of these applications, the order of observations matters. For example, the sentence “I had cleaned my car” can be changed to “I had my car cleaned,” meaning that you arranged someone else to do the work. This temporal dependence is even stronger in spoken language since several words can share a very similar sound. For example, “wreck a nice beach” sounds like “recognize speech” and the words can only be reconstructed from context.

If you think about feed-forward neural networks (including convnets) from this perspective, they seem quite limited. Those networks process incoming data in a single forward-pass, like a reflex. These networks assume all of their inputs being independent missing out many patterns in the data. While it is possible to pass inputs equal length and feed the whole sequence into the network, this does not capture the nature of sequences very well.

Recurrent Neural Networks (RNN) are a family of networks that explicitly model time. RNNs build on the same neurons summing up weighted inputs from other neurons. However, neurons are allowed to connect both forward to higher layers and backward to lower layers and form cycles. The hidden activations of the network are remembered between inputs of the same sequence.



Feed Forward Network and Recurrent Network

Various variants of RNNs have been around since the 1980s but were not widely used until recently because of insufficient computation power and difficulties in training. Since the invention of architectures like LSTM in 2006 we have seen very powerful applications of RNNs. They work well for sequential tasks in many domains, for example speech

recognition, speech synthesis, connected handwriting recognition, time-series forecasting, image caption generation, and end-to-end machine translation.

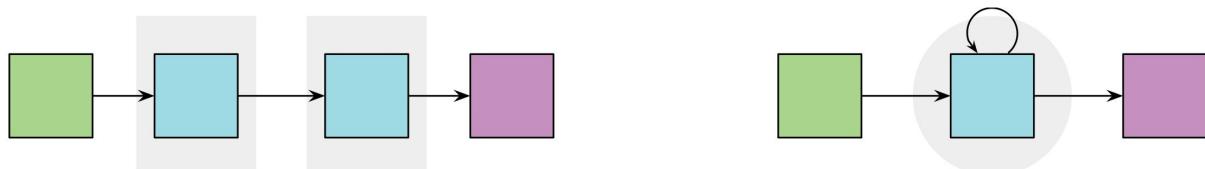
In the following sections of this chapter, we first take an in-depth look at RNNs and how to optimize them, including the required mathematical background. We then introduce variations of RNNs that help overcome certain limitations. With those tools at hand, we dive into four natural language processing tasks and apply RNNs to them. We will walk through all steps of the tasks including data handling, model design, implementation, and training in TensorFlow.

Approximating Arbitrary Programs

Let's start by introducing RNNs and gaining some intuition. Previously introduced feed-forward networks operate on fixed size vectors. For example, they map the pixels of 28x28 image to the probabilities of 10 possible classes. The computation happens in a fixed number of steps, namely the number of layers. In contrast, recurrent networks can operate on variable length sequences of vectors, either as input, output or both.

RNNs are basically arbitrary directed graphs of neurons and weights. *Input neurons* have no incoming connections because their activation is set by the input data anyway. The *output neurons* are just set of neurons in the graph that we read the prediction from. All other neurons in the graph are referred to as *hidden neurons*.

The computation performed by an RNN is analogous to a normal neural network. At each time step, we show the network the next frame of the input sequence by setting the input neurons. In contrast to forward networks, we cannot discard hidden activations because they serve as additional inputs at the next time step. The current hidden activations of an RNN are called *state*. At the beginning of each sequence, we usually start with an empty state, initialized to zeros.



Short Notation of FFNN and RNN

The state of an RNN depends on the current input and the previous state, which in turn depends on the input and state before that. Therefore, the state has indirect access to all previous inputs of the sequence and can be interpreted as a working memory.

Let's make an analogy to computer programs. Say we want to recognize the letters from an image of handwritten text. We would try and solve this with computer program in Python using variables, loops, conditionals. Feel free to try this, but I think it would be very hard to get this to work robustly.

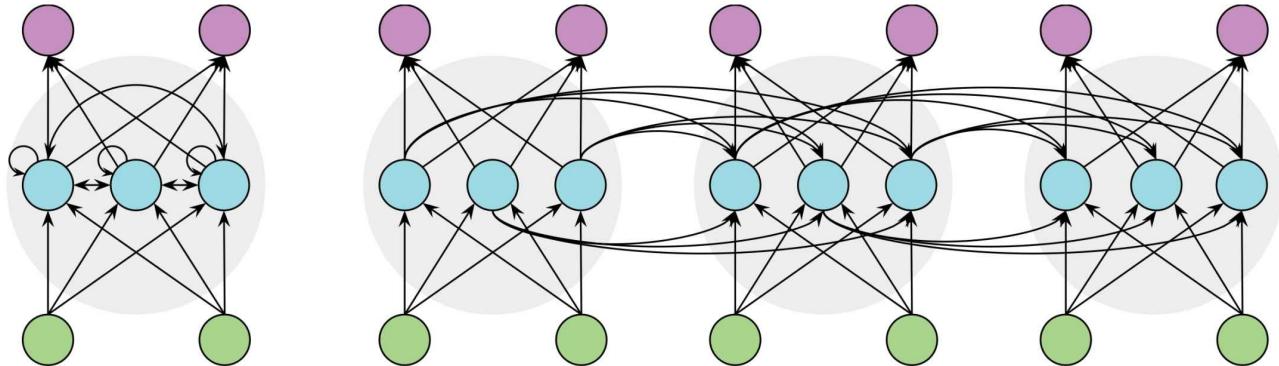
The good news is that we can train an RNN from example data instead. As we would store intermediate information in variables, the RNN learns to store intermediate information in its state. Similarly, the weight matrix of an RNN defines the program it executes, deciding what inputs to store in hidden activation and how to combine activations to new activations and outputs.

In fact, RNNs with sigmoid activations were proven to be Turing-complete by Schäfer and Zimmermann in 2006. Given the right weights, RNNs can thus compute any computable program. This is a theoretical property since there is no method to find the perfect weights for a task. However, we can already get very good results using gradient descent, as described in the next section.

Before we look into optimizing RNNs, you might ask why we even need RNNs if we can write Python programs instead? Well, the space of possible weight matrices is much easier to explore automatically than the space of possible C programs.

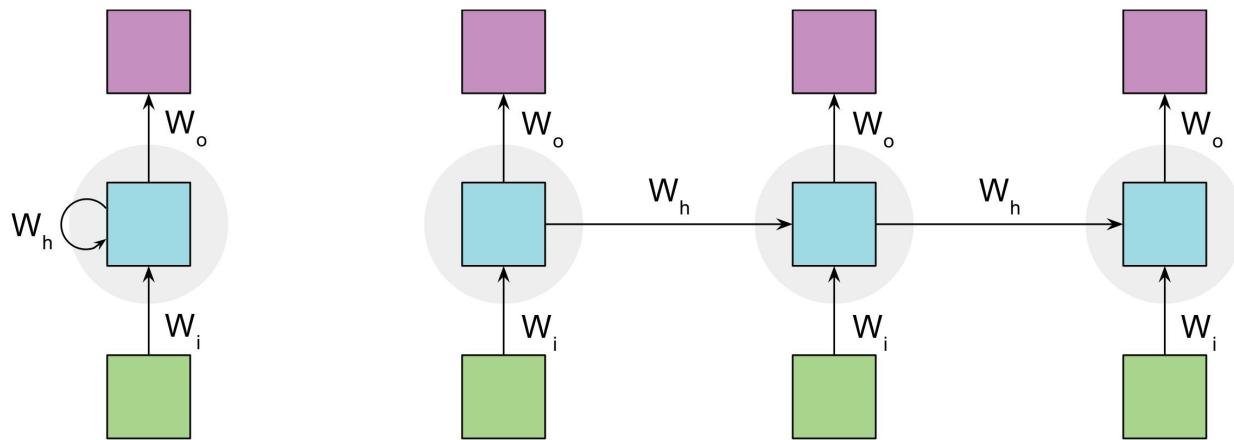
Backpropagation Through Time

Now that we have an idea of what an RNN is and why it is a cool architecture, let's take a look at how to find a good weight matrix, or how to optimize the weights. As with forward networks, the most popular optimization method is based on Gradient Descent. However, it is not straight-forward how to backpropagate the error in this dynamic system.



Unfolding a Recurrent Network in Time

The trick for optimizing RNNs is that we can unfold them in time (also referred to as unrolling) to optimize them the same way we optimize forward networks. Let's say we want to operate on sequence of length ten. We can then copy the hidden neurons ten times spanning their connections from one copy to the next one. By doing this, we get rid of recurrent connections without changing the semantics of the computation. This yields a forward network now, with the corresponding weights between time steps being tied to the same strengths. Unfolding an RNN in time does not change the computation, it is just another view.

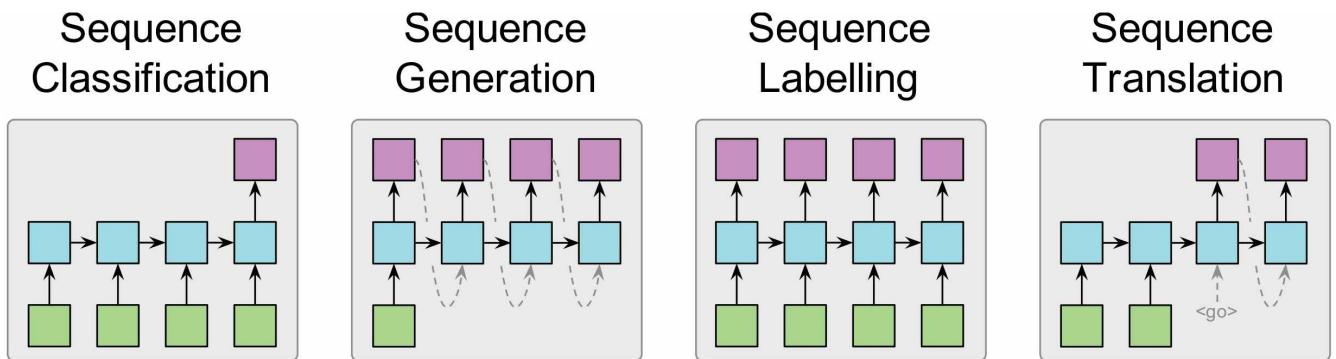


Unfolding an RNN in Time in Short Notation

We can now apply standard backpropagation through this unfolded RNN in order to compute the gradient of the error with respect to the weights. This algorithm is called *Back-Propagation Through Time* (BPTT). It will return a derivative for each weight in time, including those that are tied together. To keep tied weights at the same value, we handle them as tied weights are normally handled, that is, summing up their gradients. Note that this equals the way convolutional filters are handled in convnets.

Encoding and Decoding Sequences

The unfolded view of RNNs from the last chapter is not only useful for optimization. It also provides an intuitive way for visualizing RNNs and their input and output data. Before we get to the implementation, we will take a quick look at what mappings RNNs can perform. Sequential tasks can come in several forms: Sometimes, the input is a sequence and the output is a single vector, or the other way around. RNNs can handle those and more complicated cases well.



Common Mappings with Recurrent Networks

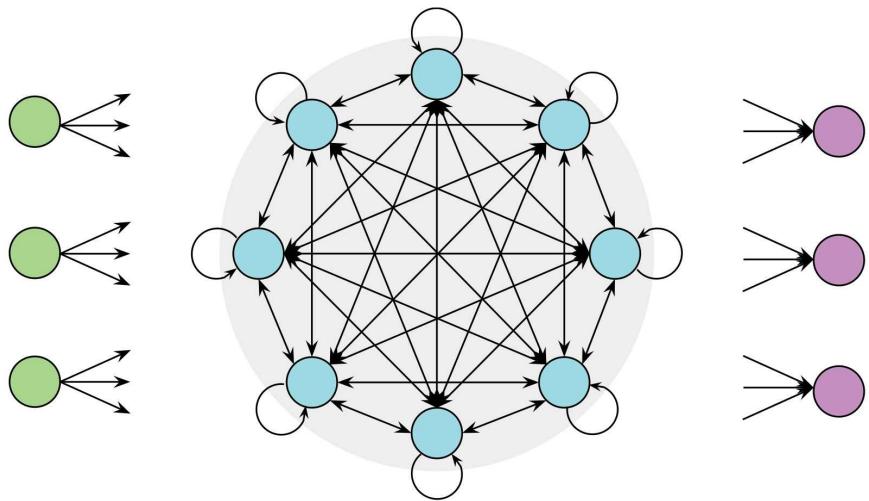
Sequence labelling is the case you probably thought of during the earlier sections. We have sequences as input and train the network to produce the right output for each frame. We are basically mapping from one sequence to another sequence of the same length.

In the *sequence classification* setting, we have sequential inputs that each have a class. We can train RNNs in this setting by only selecting the output at the last time frame. During optimization, the errors will flow back through all time steps to update the weights in order to collect and integrate useful information at each time step.

Sequence generation is the opposite case where we have a single starting point, for example a class label, that we want to generate sequences from. To generate sequences, we feed the output back into the network as next input. This makes sense since the actual output is often different from the neural network output. For example, the network might output a distribution over all classes but we only choose the most likely one.

In both sequence classification and sequence generation, we can see the single vector as dense representations of information. In first case, we encode the sequence into a dense vector to predict a class. In the second case, we decode a dense vector back into a sequence.

We can combine these approaches for *sequence translation* where we first encode a sequence of one domain, for example English. We then decode the last hidden activation back into a sequence of another domain, for example French. This works with a single RNN but when input and output are conceptually different, it can make sense to use two different RNNs and initialize the second one with the last activation of the first one. When using a single network, we need to pass a special token as input after the sequence so that the network can learn when it should stop encoding and start decoding.



Recurrent Network with Output Projection

Most often, we will use a network architecture called RNNs with *output projections*. This is an RNN with fully-connected hidden units and inputs and output mapping to or from them, respectively. Another way to look at this is that we have an RNN where all hidden units are outputs and another feed-forward layer stacked on top. You will see that this is how we implement RNNs in TensorFlow because it both is convenient and allows us to specify different activation functions to the hidden and output units.

Implementing Our First Recurrent Network

Let's implement what we have learned so far. TensorFlow supports various variants of RNNs that can be found in the `tf.nn.rnn_cell` module. With the `tf.nn.dynamic_rnn()` operation, TensorFlow also implements the RNN dynamics for us.

There is also a version of this function that adds the unfolded operations to the graph instead of using a loop. However, this consumes considerably more memory and has no real benefits. We therefore use the newer `dynamic_rnn()` operation.

As parameters, `dynamic_rnn()` takes a recurrent network definition and the batch of input sequences. For now, the sequences all have the same length. The function creates the required computations for the RNN to the compute graph and returns two tensors holding the outputs and hidden states at each time step.

```
import tensorflow as tf

# The input data has dimensions batch_size * sequence_length * frame_size.
# To not restrict ourselves to a fixed batch size, we use None as size of
# the first dimension.
sequence_length = ...
frame_size = ...
data = tf.placeholder(tf.float32, [None, sequence_length, frame_size])

num_neurons = 200
network = tf.nn.rnn_cell.BasicRNNCell(num_neurons)

# Define the operations to simulate the RNN for sequence_length steps.
outputs, states = tf.nn.dynamic_rnn(network, data, dtype=tf.float32)
```

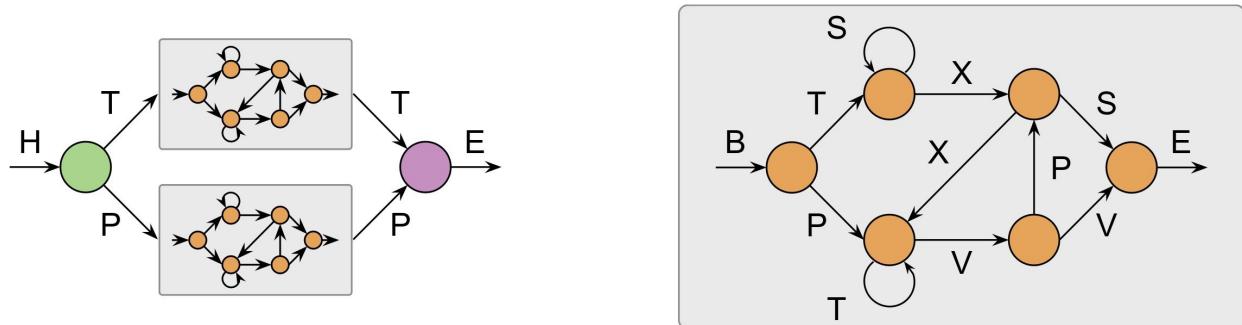
Now that we have defined the RNN and unfolded it in time, we can just load some data and train the network using one of TensorFlow's optimizers, for example

`tf.train.RMSPropOptimizer` or `tf.train.AdamOptimizer`. We will see examples of this in the later sections of this chapter where we approach practical problems with the help of RNNs.

Vanishing and Exploding Gradients

In the last section, we defined RNNs and unfolded them in time in order to backpropagate errors and apply gradient descent. However, this model would not perform very well as it stands, especially it fails to capture long-term dependencies between input frames as needed for NLP tasks for example.

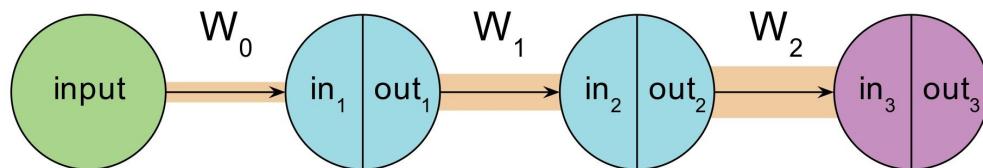
Below is an example task involving a long-term dependency where an RNN should classify whether the input sequence is part of the given grammar or not. In order to perform this task, the network has to remember the very first frame of the sequence with many unrelated frames following. Why is this a problem for the conventional RNNs we have seen so far?



A Grammar Including Long-Term Dependencies

The reason why it is difficult for an RNN to learn such long-term dependencies lies in how errors are propagated through the network during optimization. Remember that we propagate the errors through the unfolded RNN in order to compute the gradients. For long sequences, the unfolded network gets very deep and has many layers. At each layer, backpropagation scales the error from above in the network by the local derivatives.

If most of local derivatives are much smaller than the value of one, the gradient gets scaled down at every layer causing it to shrink exponentially and eventually, *vanish*. Analogously, many local derivatives being greater than one cause the gradient to *explode*.



Unstable Gradients in Deep Networks

Let's compute the gradient of this example network with just one hidden neuron per layer in order to get a better understanding of this problem. For each layer i the local derivatives $f'(in_i) * W_i^T$ get multiplied together:

$$\frac{\partial C}{\partial in_1} = \frac{\partial C}{\partial out_3} \frac{\partial out_3}{\partial in_3} * \frac{\partial in_3}{\partial out_2} \frac{\partial out_2}{\partial in_2} * \frac{\partial in_2}{\partial out_1} \frac{\partial out_1}{\partial in_1}$$

Resolving the derivatives yields:

$$cost'(f(out_3)) * f'(in_3) * W_2^T * f'(in_2) * W_1^T * f'(in_1)$$

As you can see, the error term contains the transposed weight matrix several times as a multiplicative term. In our toy network, the weight matrix contains just one element and it's easy to see that the terms gets close to zero or infinity when most weights are smaller or larger than one. In a larger network with real weight matrices, the same problem occurs when the eigen values of the weight matrices are smaller or larger than one.

This problem actually exists in any deep networks, not just recurrent ones. However, in RNNs the connections between time steps are tied each. Therefore, local derivatives of such weights are either all lesser or all greater than one and the gradient is always scaled in the same direction for each weight in the original (not unfolded) RNN. Therefore, the problem of vanishing and exploding gradients is *more prominent in RNNs than in forward networks*.

There are a couple of problems with very small or very large gradients. With elements of the gradient close to zero or infinity, learning stagnates or diverges, respectively. Moreover, we are optimizing numerically and floating point precision comes into play distorting the gradient. This problem, also known as the *fundamental problem of deep learning* has been studied and approached by many researchers in the last years. The most popular solution is an RNN architecture called *Long-Short Term Memory* (LSTM) that we will look at in the next section.

Long-Short Term Memory

Proposed in 1997 by Hochreiter & Schmidhuber, LSTM is a special form of RNN that is designed to overcome the *vanishing and exploding gradient problem*. It works significantly better for learning long-term dependencies and has become a defacto-standard for RNNs. Since then several variations of LSTM have been proposed that are also implemented in TensorFlow and that we will highlight later in this section.

To cope with the problem of vanishing and exploding gradients, the LSTM architecture replaces the normal neurons in an RNN with so-called *LSTM cells* that have a little memory inside. Those cells are wired together as they are in a usual RNN but they have an internal state that helps to remember errors over many time steps.

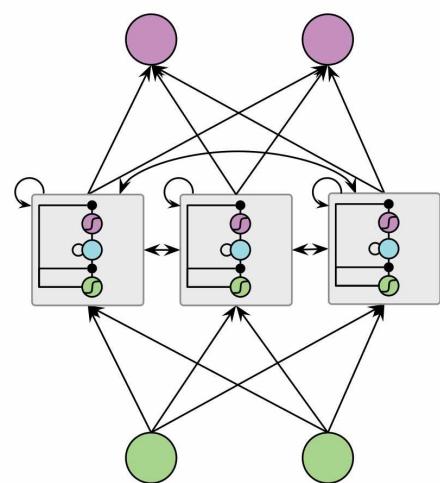
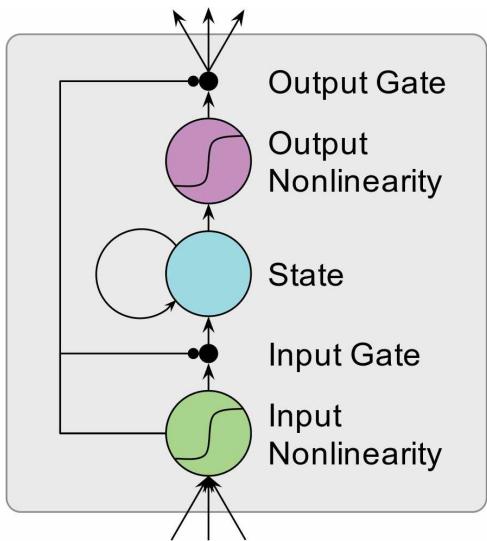
The trick of LSTM is that this internal state has a self-connection with a fixed weight of one and a linear activation function, so that its local derivative is always one. During backpropagation, this so called *constant error carousel* can carry errors over many time steps without having the gradient vanish or explode.

$$e_t = f'(in_t) * w * e_{t+1} = 1.0$$

While the purpose of the internal state is to deliver errors over many time steps, the LSTM architecture leaves learning to the *surrounding gates* that have non-linear, usually sigmoid, activation functions. In the original LSTM cell, there are two gates: One learns to scale the incoming activation and one learns to scale the outgoing activation. The cell can thus learn when to incorporate or ignore new inputs and when to release the feature it represents to other cells. The input to a cell is feeded into all gates using individual weights.

We also refer to recurrent networks as layers because we can use them as part of larger architectures. For example, we could first feed the time steps through several convolution and pooling layers, then process these outputs with an LSTM layer and add a softmax layer ontop of the LSTM activation at the last time step.

TensorFlow provides such an LSTM network with the `LSTMCell` class that can be used as a drop-in replacement for `BasicRNNCell` but also provides some additional switches. Despite its name, this class represents a whole LSTM layer. In the later sections we will see how to connect LSTM layers to other networks in order to form larger architectures.

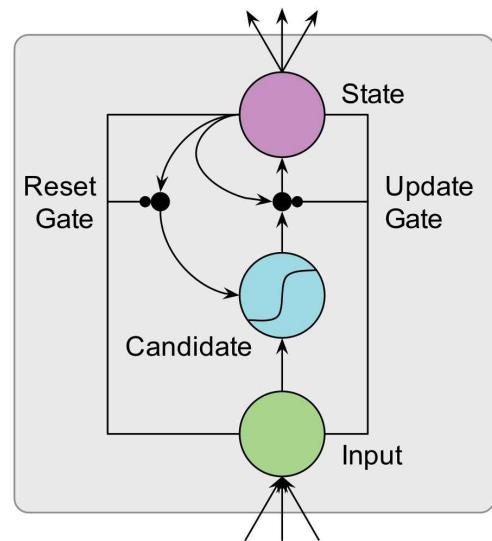
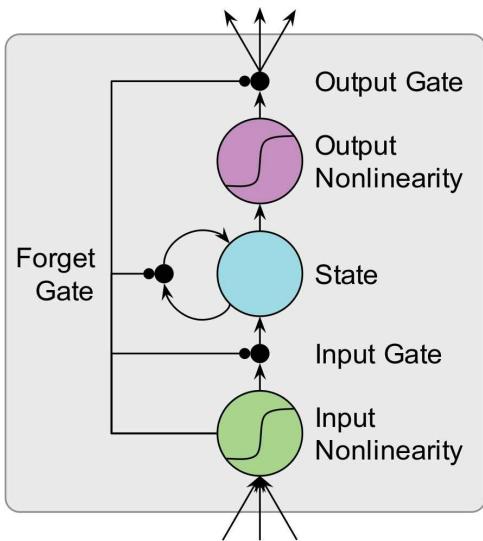


Long Short-Term Memory (LSTM)

Architecture Variations

A popular extension to LSTM is to add a *forget gate* scaling the internal recurrent connection, allowing the network to learn to forget (Gers, Felix A., Jürgen Schmidhuber, and Fred Cummins. “Learning to forget: Continual prediction with LSTM.” *Neural computation* 12.10 (2000): 2451-2471.). The derivative of the internal recurrent connection is now the activation of the forget gate and can differ from the value of one. The network can still learn to leave the forget gate closed as long as remembering the cell context is important.

It is important to initialize the forget gate to a value of one so that the cell starts in a remembering state. Forget gates are the default in almost all implementations nowadays. In TensorFlow, we can initialize the bias values of the forget gates by specifying the `forget_bias` parameter to the LSTM layer. The default is the value one and usually it's best to leave it that way.

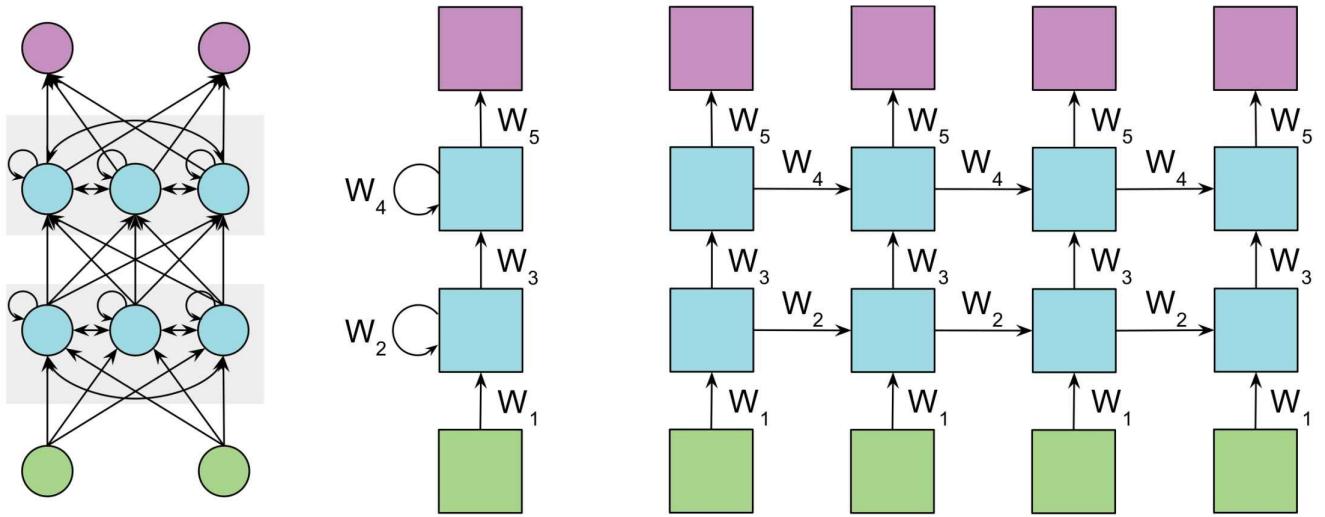


LSTM with Forget Gate and Gated Recurrent Unit (GRU)

Another extension are so called *peephole connections*, which allows the gates to look at the cell state (Gers, Felix A., Nicol N. Schraudolph, and Jürgen Schmidhuber. “Learning precise timing with LSTM recurrent networks.” *The Journal of Machine Learning Research* 3 (2003): 115-143.). The authors claim that peephole connections are beneficial when the task involves precise timing and intervals. TensorFlow’s LSTM layer supports peephole connections. They can be activated by passing the `use_peepholes=True` flag to the LSTM layer.

Based on the idea of LSTM, an alternative memory cell called *Gated Recurrent Unit* (GRU) has been proposed in 2014 (Chung, Junyoung, et al. “Empirical evaluation of gated recurrent neural networks on sequence modeling.” *arXiv preprint arXiv:1412.3555* (2014).). In contrast to LSTM, GRU has a simpler architecture and requires less computation while yielding very similar results. GRU has no output gate and combines the input and forget gates into a single *update gate*.

This update gate determines how much the internal state is blended with a candidate activation. The candidate activation is computed from a fraction of the hidden state determined by the so-called *reset gate* and the new input. The TensorFlow GRU layer is called `GRUCell` and have no parameters other than the number of cells in the layer. For further reading, we suggest the 2015 paper by Jozefowicz et al. who empirically explored recurrent cell architectures (Jozefowicz, Rafal, Wojciech Zaremba, and Ilya Sutskever. “An empirical exploration of recurrent network architectures.” *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 2015.).



Multiple Layers in Recurrent Networks

So far we looked at RNNs with fully connected hidden units. This is the most general architecture since the network can learn to set unneeded weights to zero during training. However, it is common to stack two or more layers of fully-connected RNNs on top of each other. This can still be seen as one RNN that has some structure in its connections. Since information can only flow upward between layers, multi-layer RNNs have less weights than a huge fully connected RNN and tend to learn more abstract features.

Word Vector Embeddings

In this section we will implement a model to learn word embeddings, a very powerful way to represent words for NLP tasks. The topic of word vector embeddings has recently gained popularity since methods became efficient enough to run on large text corpora. We do not use an RNN for this task yet but we will rely on this section in all further tasks. If you are familiar with the concept of word vectors and tools like word2vec but are not interested in implementing it yourself, feel free to skip ahead to the next section.

Why to represent words as vectors? The most straight-forward way to feed words into a learning system is *one-hot encoded*, that is, as a vector of the vocabulary size with all elements zero except for the position of that word set to one. There are two problems with this approach: First, the vectors are very long for real applications since there are many different words in a natural language. Second, this one-hot representation does not provide any semantic relatedness between words that certainly exists.

1	0	0	0	0	0	0	0	King
0	1	0	0	0	0	0	0	Queen
0	0	1	0	0	0	0	0	Princess

One-Hot Encoded Representation of Words

As a solution to the semantic relatedness, the idea of representing words by their cooccurrences has been around for a long time. Basically, we run over a large corpus of text and for each word count the surrounding words within a distance of, say, five. Each word is then represented by the normalized counts of nearby words. The idea behind this is that words that are used in similar contexts are similar in a semantic way. We could then compress the occurrence vectors to fewer dimensions by applying PCA or similar a similar method to get denser representations. While this approach leads to quite good performance, it requires us to keep track of the whole cooccurrence that is a square matrix of the size of our vocabulary.



Distributed Representation of Words

In 2013, Mikolov et al. came up with a practical and efficient way to compute word representations from context. The paper is: *Mikolov, Tomas, et al. “Efficient estimation of word representations in vector space.” arXiv preprint arXiv:1301.3781 (2013)*. Their *skip-gram model* starts with random representations and has a simple classifier that tries to predict a context word from the current word. The errors are propagated through both the classifier weights and the word representations and we adjust both to reduce the prediction error. It has been found that training this model over a large corpus makes the representation vectors approximate compressed co-occurrence vectors. We will now implement the skip-gram model in TensorFlow.

Preparing the Wikipedia Corpus

Before going into details of the skip-gram model, we prepare our dataset, an English Wikipedia dump in this case. The default dumps contain the full revision history of all pages but we already have enough data with the about 100GB of text from current page versions. This exercise also works for other languages and you can access an overview of the available dumps at the Wikimedia Downloads website:

<https://dumps.wikimedia.org/backup-index.html>.

```
import bz2
import collections
import os
import re

class Wikipedia:

    def __init__(self, url, cache_dir, vocabulary_size=10000):
        pass

    def __iter__(self):
        """Iterate over pages represented as lists of word indices."""
        pass

    @property
    def vocabulary_size(self):
        pass

    def encode(self, word):
        """Get the vocabulary index of a string word."""
        pass

    def decode(self, index):
        """Get back the string word from a vocabulary index."""
        pass

    def _read_pages(self, url):
        """
        Extract plain words from a Wikipedia dump and store them to the pages
        file. Each page will be a line with words separated by spaces.
        """
        pass

    def _build_vocabulary(self, vocabulary_size):
        """
        Count words in the pages file and write a list of the most frequent
        words to the vocabulary file.
        """
        pass

    @classmethod
    def _tokenize(cls, page):
        pass
```

There are a couple of steps to perform in order to get the data into the right format. As you might have seen earlier in this book, data collection and cleaning is both a demanding and important task. Ultimately, we would like to iterate over Wikipedia pages represented as one-hot encoded words. We do this in multiple steps:

1. Download the dump and extract pages and their words.
2. Count words to form a vocabulary of the most common words.
3. Encode the extracted pages using the vocabulary.

The whole corpus does not fit into main memory easily, so we have to perform these

operations on data streams by reading the file line by line and write the intermediate results back to disk. This way, we have checkpoints between the steps so that we don't have to start all over if something crashes. We use the following class to handle the Wikipedia processing. In the `__init__()` you can see the checkpointing logic using file-existance checks.

```

def __init__(self, url, cache_dir, vocabulary_size=10000):
    self._cache_dir = os.path.expanduser(cache_dir)
    self._pages_path = os.path.join(self._cache_dir, 'pages.bz2')
    self._vocabulary_path = os.path.join(self._cache_dir, 'vocabulary.bz2')
    if not os.path.isfile(self._pages_path):
        print('Read pages')
        self._read_pages(url)
    if not os.path.isfile(self._vocabulary_path):
        print('Build vocabulary')
        self._build_vocabulary(vocabulary_size)
    with bz2.open(self._vocabulary_path, 'rt') as vocabulary:
        print('Read vocabulary')
        self._vocabulary = [x.strip() for x in vocabulary]
    self._indices = {x: i for i, x in enumerate(self._vocabulary)}

def __iter__(self):
    """Iterate over pages represented as lists of word indices."""
    with bz2.open(self._pages_path, 'rt') as pages:
        for page in pages:
            words = page.strip().split()
            words = [self.encode(x) for x in words]
            yield words

@property
def vocabulary_size(self):
    return len(self._vocabulary)

def encode(self, word):
    """Get the vocabulary index of a string word."""
    return self._indices.get(word, 0)

def decode(self, index):
    """Get back the string word from a vocabulary index."""
    return self._vocabulary[index]

```

As you have noticed, we still have to implement two important functions of this Wclass. The first one, `_read_pages()` will download the Wikipedia dump which comes as a compressed XML file, iterate over the pages and extract the plain words to get rid of any formatting. To read the compressed file, we need the `bz2` module that provides an `open()` function that works similar to its standard equivalent but takes care of compression and decompression, even when streaming the file. To save some disk space, we will also use this compression for the intermediate results. The regex used to extract words just captures any sequence of consecutive letter and individual occurrences of some special characters.

```

from lxml import etree

TOKEN_REGEX = re.compile(r'[A-Za-z]+|[!?.:,()]+')

def _read_pages(self, url):
    """
    Extract plain words from a Wikipedia dump and store them to the pages
    file. Each page will be a line with words separated by spaces.
    """
    wikipedia_path = download(url, self._cache_dir)
    with bz2.open(wikipedia_path) as wikipedia, \
        bz2.open(self._pages_path, 'wt') as pages:
        for _, element in etree.iterparse(wikipedia, tag='{*}page'):
            if element.find('./{*}redirect') is not None:
                continue
            page = element.findtext('./{*}revision/{*}text')
            words = self._tokenize(page)

```

```

    pages.write(' '.join(words) + '\n')
    element.clear()

@classmethod
def _tokenize(cls, page):
    words = cls.TOKEN_REGEX.findall(page)
    words = [x.lower() for x in words]
    return words

```

We need a vocabulary of words to use for the one-hot encoding. We can then encode each word by its index in the vocabulary. To remove some misspelled or very uncommon words, the vocabulary only contains the the `vocabulary_size - 1` most common words and an `<unk>` token that will be used for every word that is not in the vocabulary. This token will also give us a word-vector that we can use for unseen words later.

```

def _build_vocabulary(self, vocabulary_size):
    """
    Count words in the pages file and write a list of the most frequent
    words to the vocabulary file.
    """
    counter = collections.Counter()
    with bz2.open(self._pages_path, 'rt') as pages:
        for page in pages:
            words = page.strip().split()
            counter.update(words)
    common = ['<unk>'] + counter.most_common(vocabulary_size - 1)
    common = [x[0] for x in common]
    with bz2.open(self._vocabulary_path, 'wt') as vocabulary:
        for word in common:
            vocabulary.write(word + '\n')

```

Since we extracted the plain text and defined the encoding for the words, we can form training examples of it on the fly. This is nice since storing the examples would require a lot of storage space. Most of the time will be spent for the training anyway, so this doesn't impact performance by much. We also want to group the resulting examples into batches to train them more efficiently. We will be able to use very large batches with this model because the classifier does not require a lot of memory.

So how do we form the training examples? Remember that the *skip-gram model* predicts context words from current words. While iterating over the text, we create training examples with the current word as data and its surrounding words as targets. For a context size of $R = 5$, we would thus generate ten training examples per word, with the five words to the left and right being the targets. However, one can argue that close neighbors are more important to the semantic context than far neighbors. We thus create less training examples with far context words by randomly choosing a context size in range $[1, D = 10]$ at each word.

```

def skipgrams(pages, max_context):
    """
    Form training pairs according to the skip-gram model.
    """
    for words in pages:
        for index, current in enumerate(words):
            context = random.randint(1, max_context)
            for target in words[max(0, index - context): index]:
                yield current, target
            for target in words[index + 1: index + context + 1]:
                yield current, target

def batched(iterator, batch_size):
    """
    Group a numerical stream into batches and yield them as Numpy arrays.
    """
    while True:
        data = np.zeros(batch_size)

```

```
target = np.zeros(batch_size)
for index in range(batch_size):
    data[index], target[index] = next(iterator)
yield data, target
```

Model structure

Now that we got the Wikipedia corpus prepared, we can define a model to compute the word embeddings.

```
class EmbeddingModel:  
  
    def __init__(self, data, target, params):  
        self.data = data  
        self.target = target  
        self.params = params  
        self.embeddings  
        self.cost  
        self.optimize  
  
    @lazy_property  
    def embeddings(self):  
        pass  
  
    @lazy_property  
    def optimize(self):  
        pass  
  
    @lazy_property  
    def cost(self):  
        pass
```

Each word starts off being represented by a random vector. From the intermediate representation of a word, a classifier will then try to predict the current representation of one of its context words. We will then propagate the errors to tweak both the weights and the representation of the input word. The thus use a `tf.Variable` for the representations.

```
@lazy_property  
def embeddings(self):  
    initial = tf.random_uniform(  
        [self.params.vocabulary_size, self.params.embedding_size],  
        -1.0, 1.0)  
    return tf.Variable(initial)
```

We use the `MomentumOptimizer` that is not very clever but has the advantage of being very fast. This makes it play nicely with our large Wikipedia corpus and the idea behind skip-gram to prefer more data over clever algorithms.

```
@lazy_property  
def optimize(self):  
    optimizer = tf.train.MomentumOptimizer(  
        self.params.learning_rate, self.params.momentum)  
    return optimizer.minimize(self.cost)
```

The only missing part of our model is the classifier. This is the heart of the successful *skip-gram* model and we will now take a look at how it works.

Noise Contrastive Classifier

There are multiple cost functions for the skip-gram model but one that has been found to work very well is *noise-contrastive estimation loss*. Ideally, we not only want the predictions to be close to the targets but also far from words that are not targets for the current word. This could be nicely modelled as a softmax classifier but we do not want to compute and train the outputs for all words in the alphabet every time. The idea is to always use some new random vectors as negative examples, also called contrastive examples. Over enough training iterations this averages to the softmax classifier while only requiring tens of classes. TensorFlow provides a convenient `tf.nn.nce_loss` function for this.

```
@lazy_property
def cost(self):
    embedded = tf.nn.embedding_lookup(self.embeddings, self.data)
    weight = tf.Variable(tf.truncated_normal(
        [self.params.vocabulary_size, self.params.embedding_size],
        stddev=1.0 / self.params.embedding_size ** 0.5))
    bias = tf.Variable(tf.zeros([self.params.vocabulary_size]))
    target = tf.expand_dims(self.target, 1)
    return tf.reduce_mean(tf.nn.nce_loss(
        weight, bias, embedded, target,
        self.params.contrastive_examples,
        self.params.vocabulary_size))
```

Training the model

We prepared the corpus and defined the model. Here is the remaining code to put things together. After training, we store the final embeddings into another file. The example below uses only a subset of Wikipedia that already takes about 5 hours to train on an average CPU. To use the full corpus, just switch the url to <https://dumps.wikimedia.org/enwiki/20160501/enwiki-20160501-pages-meta-current.xml.bz2>.

As you can see, we make use of a `AttrDict` class. This is equivalent to a Python `dict` except that we can access keys as if they were attributes, for example `params.batch_size`. For more details, please see the chapter *Code Structure and Utilities*.

```
params = AttrDict(
    vocabulary_size=10000,
    max_context=10,
    embedding_size=200,
    contrastive_examples=100,
    learning_rate=0.5,
    momentum=0.5,
    batch_size=1000,
)
data = tf.placeholder(tf.int32, [None])
target = tf.placeholder(tf.int32, [None])
model = EmbeddingModel(data, target, params)

corpus = Wikipedia(
    'https://dumps.wikimedia.org/enwiki/20160501/' \
    'enwiki-20160501-pages-meta-current1.xml-p000000010p000030303.bz2',
    '/home/user/wikipedia',
    params.vocabulary_size)
examples = skipgrams(corpus, params.max_context)
batches = batched(examples, params.batch_size)

sess = tf.Session()
sess.run(tf.initialize_all_variables())
average = collections.deque(maxlen=100)
for index, batch in enumerate(batches):
    feed_dict = {data: batch[0], target: batch[1]}
    cost, _ = sess.run([model.cost, model.optimize], feed_dict)
    average.append(cost)
    print('{:}: {:.1f}'.format(index + 1, sum(average) / len(average)))

embeddings = sess.run(model.embeddings)
np.save('/home/user/wikipedia/embeddings.npy', embeddings)
```

After about five hours of training, this we will get the learned embeddings as a stored Numpy array. While we will use the embeddings in the later chapter, you don't have to compute them yourself, if you don't want to. There are pre-trained word embeddings available online and we will point to them later when we need them.

Sequence Classification

Sequence classification is a problem setting where we predict a class for the whole input sequence. Such problems are common in many fields including genomics and finance. A prominent from NLP is sentiment analysis: Predicting the attitude towards a given topic from user-written text. For example, one could predict the sentiment of tweets mentioning a certain candidate in an election and use that to forecast the election results. Another example is predicting product or movie ratings from written reviews. This is used as a benchmark task in the NLP community because reviews often contain numerical ratings that make for convenient target values.

We will use a dataset of movie reviews from the *International Movie Database* with the binary targets *positive* or *negative*. On this dataset, native methods that just look at the existence of words tend to fail because of negations, irony and ambiguity in language in general. We will build a recurrent model operating on the word vectors from the last section. The recurrent network will see a review word-by-word. From the *activation at the last word*, we will train a classifier to predict the sentiment of the whole review. Because we train the architecture end-to-end, the RNN will collect and encode the useful information from the words that will be most valuable for the later classification.

Imdb Movie Review Dataset

The movie review dataset is offered by Stanford University's AI department: <http://ai.stanford.edu/~amaas/data/sentiment/>. It comes as a compressed `tar` archive where positive and negative reviews can be found as text files in two according folders. We apply the same pre-processing to the text as in the last section: Extracting plain words using a regular expression and converting to lower case.

```
import tarfile
import re

class ImdbMovieReviews:

    DEFAULT_URL = \
        'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz'
    TOKEN_REGEX = re.compile(r'[A-Za-z]+|[!?.:,()]+')

    def __init__(self, cache_dir, url=None):
        self._cache_dir = cache_dir
        self._url = url or type(self).DEFAULT_URL

    def __iter__(self):
        filepath = download(self._url, self._cache_dir)
        with tarfile.open(filepath) as archive:
            for filename in archive.getnames():
                if filename.startswith('aclImdb/train/pos/'):
                    yield self._read(archive, filename), True
                elif filename.startswith('aclImdb/train/neg/'):
                    yield self._read(archive, filename), False

    def _read(self, archive, filename):
        with archive.extractfile(filename) as file_:
            data = file_.read().decode('utf-8')
            data = type(self).TOKEN_REGEX.findall(data)
            data = [x.lower() for x in data]
        return data
```

Using the Word Embeddings

As explained in the *Word Vector Embeddings* section, embeddings are semantically richer than one-hot encoded words. We can help our RNN by letting it operate on the embedded words of movie reviews rather than one-hot encoded words. For this, we will use the vocabulary and embeddings that we computed in the referenced section. The code should be straight forward. We just use the vocabulary to determine the index of a word and use that index to find the right embedding vector. The following class also pads the sequences to the same length so we can easily fit batches of multiple reviews into your network later.

```
import bz2
import numpy as np

class Embedding:

    def __init__(self, vocabulary_path, embedding_path, length):
        self._embedding = np.load(embedding_path)
        with bz2.open(vocabulary_path, 'rt') as file_:
            self._vocabulary = {k.strip(): i for i, k in enumerate(file_)}
        self._length = length

    def __call__(self, sequence):
        data = np.zeros((self._length, self._embedding.shape[1]))
        indices = [self._vocabulary.get(x, 0) for x in sequence]
        embedded = self._embedding[indices]
        data[:len(sequence)] = embedded
        return data

    @property
    def dimensions(self):
        return self._embedding.shape[1]
```

Sequence Labelling Model

We want to classify the sentiment of text sequences. Because this is a supervised setting, we pass two placeholders to the model: one for the input `data`, or the sequence, and one for the `target` value, or the sentiment. We also pass in the `params` object that contains configuration parameters like the size of the recurrent layer, its cell architecture (LSTM, GRU, etc), and the optimizer to use. We will now implement the properties and discuss them in detail.

```
class SequenceClassificationModel:
```

```
    def __init__(self, data, target, params):
        self.data = data
        self.target = target
        self.params = params
        self.prediction
        self.cost
        self.error
        self.optimize

    @lazy_property
    def length(self):
        pass

    @lazy_property
    def prediction(self):
        pass

    @lazy_property
    def cost(self):
        pass

    @lazy_property
    def error(self):
        pass

    @lazy_property
    def optimize(self):
        pass

    @staticmethod
    def _last_relevant(output, length):
        pass
```

First, we obtain the *lengths of sequences* in the current data batch. We need this since the data comes as a single tensor, padded with zero vectors to the longest review length. Instead of keeping track of the sequence lengths of every review, we just compute it dynamically in TensorFlow. To get the length per sequence, we collapse the word vectors using the maximum on the absolute values. The resulting scalars will be zero for zero vectors and larger than zero for any real word vector. We then discretize these values to zero or one using `tf.sign()` and sum up the results along the time steps to obtain the length of each sequence. The resulting tensor has the length of batch size and contains a scalar length for each sequence.

```
@lazy_property
def length(self):
    used = tf.sign(tf.reduce_max(tf.abs(self.data), reduction_indices=2))
    length = tf.reduce_sum(used, reduction_indices=1)
    length = tf.cast(length, tf.int32)
    return length
```

Softmax from last relevant activation

For the prediction, we define an RNN as usual. However, this time we want to augment it by *stacking a softmax layer ontop of its last activation*. For the RNN, we use a cell type and cell count defined in the `params` object. We use the already defined `length` property to only show rows of the batch to the RNN up to their length. We can then fetch the last output activation of each sequence and feed that into a softmax layer. Defining the softmax layer should be pretty straight forward if you've followed the book up to this section.

Note that the last relevant output activation of the RNN has a different index for each sequence in the training batch. This is because each review has a different length. We already know the length of each sequence, so how do we use that to select the last activations? The problem is that we want to index in the dimension of time steps, which is the second dimension in the batch of shape `sequences x time_steps x word_vectors`.

```
@lazy_property
def prediction(self):
    # Recurrent network.
    output, _ = tf.nn.dynamic_rnn(
        self.params.rnn_cell(self.params.rnn_hidden),
        self.data,
        dtype=tf.float32,
        sequence_length=self.length,
    )
    last = self._last_relevant(output, self.length)
    # Softmax layer.
    num_classes = int(self.target.get_shape()[1])
    weight = tf.Variable(tf.truncated_normal(
        [self.params.rnn_hidden, num_classes], stddev=0.01))
    bias = tf.Variable(tf.constant(0.1, shape=[num_classes]))
    prediction = tf.nn.softmax(tf.matmul(last, weight) + bias)
    return prediction
```

As of now, TensorFlow only supports indexing along the first dimension, using `tf.gather()`. We thus flatten the first two dimensions of the output activations from their shape of `sequences x time_steps x word_vectors` and construct an index into this resulting tensor. The index takes into account the start indices for each sequence in the flat tensor and adds the sequence length to it. Actually, we only add `length - 1` so that we select the last valid time step.

```
@staticmethod
def _last_relevant(output, length):
    batch_size = tf.shape(output)[0]
    max_length = int(output.get_shape()[1])
    output_size = int(output.get_shape()[2])
    index = tf.range(0, batch_size) * max_length + (length - 1)
    flat = tf.reshape(output, [-1, output_size])
    relevant = tf.gather(flat, index)
    return relevant
```

We will be able to train the whole model end-to-end with TensorFlow propagating the errors through the softmax layer and the used time steps of the RNN. The only thing that is missing for training is a cost function.

Gradient clipping

For sequence classification, we can use any cost function that makes sense for classification because the model output is just a probability distribution over the available classes. In our example, the two classes are *positive* and *negative* sentiment and we will use a standard cross-entropy cost as explain in the previous chapter on object recognition and classification.

To minimize the cost function, we use the optimizer defined in the configuration. However, we will improve on what we've learned so far by adding *gradient clipping*. RNNs are quite hard to train and weights tend to diverge if the hyper parameters do not play nicely together. The idea of gradient clipping is to restrict the the values of the gradient to a sensible range. This way, we can limit the maximum weight updates.

```
@lazy_property
def cost(self):
    cross_entropy = -tf.reduce_sum(self.target * tf.log(self.prediction))
    return cross_entropy

@lazy_property
def optimize(self):
    gradient = self.params.optimizer.compute_gradients(self.cost)
    if self.params.gradient_clipping:
        limit = self.params.gradient_clipping
        gradient = [
            (tf.clip_by_value(g, -limit, limit), v)
            if g is not None else (None, v)
            for g, v in gradient]
    optimize = self.params.optimizer.apply_gradients(gradient)
    return optimize

@lazy_property
def error(self):
    mistakes = tf.not_equal(
        tf.argmax(self.target, 1), tf.argmax(self.prediction, 1))
    return tf.reduce_mean(tf.cast(mistakes, tf.float32))
```

TensorFlow supports this szenario with the `compute_gradients()` function that each optimizer instance provides. We can then modify the gradients and apply the weight changes with `apply_gradients()`. For gradient clipping, we set elements to `-limit` if they are lower than that or to `limit` if they are larger than that. The only tricky part is that derivatives in TensorFlow can be `None` which means there is no relation between a variable and the cost function. Mathematically, those derivatives should be zero vectors, but using `None` allows for internal performance optimizations. We handle those cases by just passing the `None` value back as in the tuple.

Training the model

Let's now train the advanced model we defined above. As we said, we will feed the movie reviews into the recurrent network word-by-word so each time step is a batch of word vectors. We adapt the `batched()` function from the last section to lookup the word vectors and padd all sequences to the same length as follows:

```
def preprocess_batched(iterator, length, embedding, batch_size):
    iterator = iter(iterator)
    while True:
        data = np.zeros((batch_size, length, embedding.dimensions))
        target = np.zeros((batch_size, 2))
        for index in range(batch_size):
            text, label = next(iterator)
            data[index] = embedding(text)
            target[index] = [1, 0] if label else [0, 1]
        yield data, target
```

We can easily train the model now. We define the hyper parameters, load the dataset and embeddings, and run the model on the preprocessed training batches.

```
params = AttrDict(
    rnn_cell=GRUCell,
    rnn_hidden=300,
    optimizer=tf.train.RMSPropOptimizer(0.002),
    batch_size=20,
)

reviews = ImdbMovieReviews('/home/user/imdb')
length = max(len(x[0]) for x in reviews)

embedding = Embedding(
    '/home/user/wikipedia/vocabulary.bz2',
    '/home/user/wikipedia/embedding.npy', length)
batches = preprocess_batched(reviews, length, embedding, params.batch_size)

data = tf.placeholder(tf.float32, [None, length, embedding.dimensions])
target = tf.placeholder(tf.float32, [None, 2])
model = SequenceClassificationModel(data, target, params)

sess = tf.Session()
sess.run(tf.initialize_all_variables())
for index, batch in enumerate(batches):
    feed = {data: batch[0], target: batch[1]}
    error, _ = sess.run([model.error, model.optimize], feed)
    print('{:}: {:.3f}'.format(index + 1, 100 * error))
```

This time, the training success of this model will not only depend on the network structure and hyper parameter, but also on the quality of the word embeddings. If you did not train your own word embeddings as described in the last section, you can load pre-trained embeddings from the word2vec project:

<https://code.google.com/archive/p/word2vec/> that implements the skip-gram model, or the very similar Glove model from the Stanford NLP group:

<http://nlp.stanford.edu/projects/glove/>. In both cases you will be able to find Python loaders on the web.

We have this model now, so what can you do with it? There is an open learning competition on Kaggle, a famous website hosting data science challenges. It uses the same IMDB movie review dataset as we did in this section. So if you are interested how your results compare to others, you can run the model on their testset and upload your results at <https://www.kaggle.com/c/word2vec-nlp-tutorial>.

Sequence Labelling

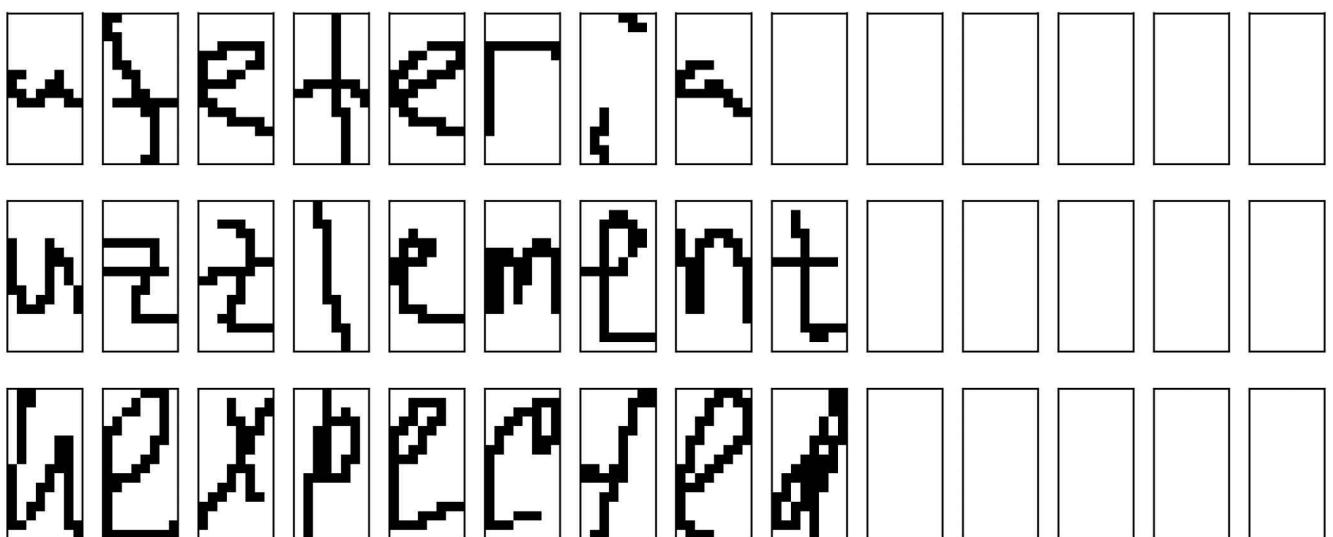
In the last section, we built a sequence classification model that uses an LSTM network and stacked a softmax layer ontop of the last activation. Building on this, we will now tackle the slightly harder problem of *sequence labelling*. This setting differs from sequence classification in that we want to predict an individual class for each frame in the input sequence.

For example, let's think about regocnizing handwritten text. Each word is a sequence of letters and while we could classify each letter independently, human language has a strong structure that we can take advantage of. If you take a look at a handwritten sample, there are often letters that are hard to read on their own, for example “n”, “m”, and “u”. They can be regocnized from the context of nearby letters however. In this section, we will use RNNs to make use of the dependencies between letters and built a more robust OCR (Optical Character Recognition) system.

Optical Character Recognition Dataset

As an example of sequence labelling problems, we will take a look at the OCR dataset collected by Rob Kassel at the MIT Spoken Language Systems Group and preprocessed by Ben Taskar at the Stanford AI group. The dataset contains individual hand-written letters as binary images of 16 times 8 pixels. The letters are arranged into sequences that where each sequence forms a word. In the whole dataset, there are about 6800 words of length up to 14.

Here are three example sequence from the OCR dataset. The words are “cafeteria”, “puzzlement”, and “unexpected”. The first letters are not included in the dataset since they were uppercase. All sequences are padded to maximal length of 14. To make it a little easier, the dataset contains only lowercase letters. This is why some words miss their first letter.



The dataset is available at <http://ai.stanford.edu/~btaskar/ocr/> and comes as a gzipped tab separated textfile that we can read using Python’s `csv` module. Each line represents a letter of the dataset and holds attributes like and id, the target letter, the pixel values and the id of the following letter of the word.

```
import gzip
import csv
import numpy as np

class OcrDataset:
    """
    Dataset of handwritten words collected by Rob Kassel at the MIT Spoken
    Language Systems Group. Each example contains the normalized letters of the
    word, padded to the maximum word length. Only contains lower case letter,
    capitalized letters were removed.
    From: http://ai.stanford.edu/~btaskar/ocr/
    """

    URL = 'http://ai.stanford.edu/~btaskar/ocr/letter.data.gz'

    def __init__(self, cache_dir):
        path = download(type(self).URL, cache_dir)
        lines = self._read(path)
        data, target = self._parse(lines)
        self.data, self.target = self._pad(data, target)
```

```

@staticmethod
def _read(filepath):
    with gzip.open(filepath, 'rt') as file_:
        reader = csv.reader(file_, delimiter='\t')
        lines = list(reader)
    return lines

@staticmethod
def _parse(lines):
    lines = sorted(lines, key=lambda x: int(x[0]))
    data, target = [], []
    next_ = None
    for line in lines:
        if not next_:
            data.append([])
            target.append([])
        else:
            assert next_ == int(line[0])
        next_ = int(line[2]) if int(line[2]) > -1 else None
        pixels = np.array([int(x) for x in line[6:134]])
        pixels = pixels.reshape((16, 8))
        data[-1].append(pixels)
        target[-1].append(line[1])
    return data, target

@staticmethod
def _pad(data, target):
    max_length = max(len(x) for x in target)
    padding = np.zeros((16, 8))
    data = [x + ([padding] * (max_length - len(x))) for x in data]
    target = [x + ([1] * (max_length - len(x))) for x in target]
    return np.array(data), np.array(target)

```

We first sort by those following id values so that we can read the letters of each word in the correct order. Then, we continue collecting letters until the field of the next id is not set in which case we start a new sequence. After reading the target letters and their data pixels, we pad the sequences with zero images so that they fit into two big Numpy arrays containing the target letters and all the pixel data.

Softmax shared between time steps

This time, not only the data but also the target array contains sequences, one target letter for each image frame. The easiest approach to get a prediction at each frame is to augment our RNN with a softmax classifier on top of the output at each letter. This is very similar to our model for sequence classification from the last section, except that we classifier is evaluated at each frame rather than just at the last one.

```
class SequenceLabellingModel:  
  
    def __init__(self, data, target, params):  
        self.data = data  
        self.target = target  
        self.params = params  
        self.prediction  
        self.cost  
        self.error  
        self.optimize  
  
    @lazy_property  
    def length(self):  
        pass  
  
    @lazy_property  
    def prediction(self):  
        pass  
  
    @lazy_property  
    def cost(self):  
        pass  
  
    @lazy_property  
    def error(self):  
        pass  
  
    @lazy_property  
    def optimize(self):  
        pass
```

Let's implement the methods of our sequence labelling mode. First off, we again need to compute the sequence lengths. We already did this in the last section so there is not much to add here.

```
@lazy_property  
def length(self):  
    used = tf.sign(tf.reduce_max(tf.abs(self.data), reduction_indices=2))  
    length = tf.reduce_sum(used, reduction_indices=1)  
    length = tf.cast(length, tf.int32)  
    return length
```

Now, we come to the prediction, were the main different to the sequence classification model lies. There would be two ways to add a softmax layer to all frames. We could either add several different classifiers or share the same among all frames. Since classifying the third letter should not be very different from classifying the eighth letter, it makes sense to take the latter approach. This way, the classifier weights are also trained more often because each letter of the word contributes to training them.

In order to implement a shared layer in TensorFlow, we have to apply a little trick. A weight matrix of a fully-connected layer always has the dimensions `batch_size x in_size x out_size`. But we now have two input dimensions along which we want to apply the matrix, `batch_size` and `sequence_steps`.

What we can do to circumvent this problem is to flatten the input to the layer, in this case the outgoing activation of the RNN, to shape `batch_size * sequence_steps x in_size`. This way, it just looks like a large batch to the weight matrix. Of course we have to reshape the results back to unflatten them.

```

@lazy_property
def prediction(self):
    output, _ = tf.nn.dynamic_rnn(
        GRUCell(self.params.rnn_hidden),
        self.data,
        dtype=tf.float32,
        sequence_length=self.length,
    )
    # Softmax layer.
    max_length = int(self.target.get_shape()[1])
    num_classes = int(self.target.get_shape()[2])
    weight = tf.Variable(tf.truncated_normal(
        [self.params.rnn_hidden, num_classes], stddev=0.01))
    bias = tf.Variable(tf.constant(0.1, shape=[num_classes]))
    # Flatten to apply same weights to all time steps.
    output = tf.reshape(output, [-1, self.params.rnn_hidden])
    prediction = tf.nn.softmax(tf.matmul(output, weight) + bias)
    prediction = tf.reshape(prediction, [-1, max_length, num_classes])
    return prediction

```

The cost and error function change slightly compared to what we had for sequence classification. Namely, there is now a prediction-target pair for each frame in the sequence, so we have to average over that dimension as well. However, `tf.reduce_mean()` doesn't work here since it would normalize by the tensor length which is the maximum sequence length. Instead, we want to normalize by the actual sequence lengths computed earlier. Thus, we manually use `tf.reduce_sum()` and a division to obtain the correct mean.

```

@lazy_property
def cost(self):
    # Compute cross entropy for each frame.
    cross_entropy = self.target * tf.log(self.prediction)
    cross_entropy = -tf.reduce_sum(cross_entropy, reduction_indices=2)
    mask = tf.sign(tf.reduce_max(tf.abs(self.target), reduction_indices=2))
    cross_entropy *= mask
    # Average over actual sequence lengths.
    cross_entropy = tf.reduce_sum(cross_entropy, reduction_indices=1)
    cross_entropy /= tf.cast(self.length, tf.float32)
    return tf.reduce_mean(cross_entropy)

```

Analogously to the cost, we have to adjust the error function. The axis that `tf.argmax()` operates on is axis two rather than axis one now. Then we mask padding frames and compute the average over the actual sequence length. The last `tf.reduce_mean()` averages over the words in the data batch.

```

@lazy_property
def error(self):
    mistakes = tf.not_equal(
        tf.argmax(self.target, 2), tf.argmax(self.prediction, 2))
    mistakes = tf.cast(mistakes, tf.float32)
    mask = tf.sign(tf.reduce_max(tf.abs(self.target), reduction_indices=2))
    mistakes *= mask
    # Average over actual sequence lengths.
    mistakes = tf.reduce_sum(mistakes, reduction_indices=1)
    mistakes /= tf.cast(self.length, tf.float32)
    return tf.reduce_mean(mistakes)

```

The nice thing about TensorFlow's automatic gradient computation is that we can use the same optimization operation for this model as we used for sequence classification, just plugging in the new cost function. We will apply gradient clipping in all RNNs from now

on, since it can prevent divergence during training while it does not have any negative impact.

```
@lazy_property
def optimize(self):
    gradient = self.params.optimizer.compute_gradients(self.cost)
    if self.params.gradient_clipping:
        limit = self.params.gradient_clipping
        gradient = [
            (tf.clip_by_value(g, -limit, limit), v)
            if g is not None else (None, v)
            for g, v in gradient]
    optimize = self.params.optimizer.apply_gradients(gradient)
    return optimize
```

Training the Model

Now we can put together the pieces described so far and train the model. The imports and configuration parameters should be familiar to you from the previous section. We then use `get_dataset()` to download and preprocess the handwritten images. This is also where we encode the targets from lower-case letters to one-hot vectors. After the encoding, we shuffle the data so that we get unbiased splits for training and testing.

```
import random

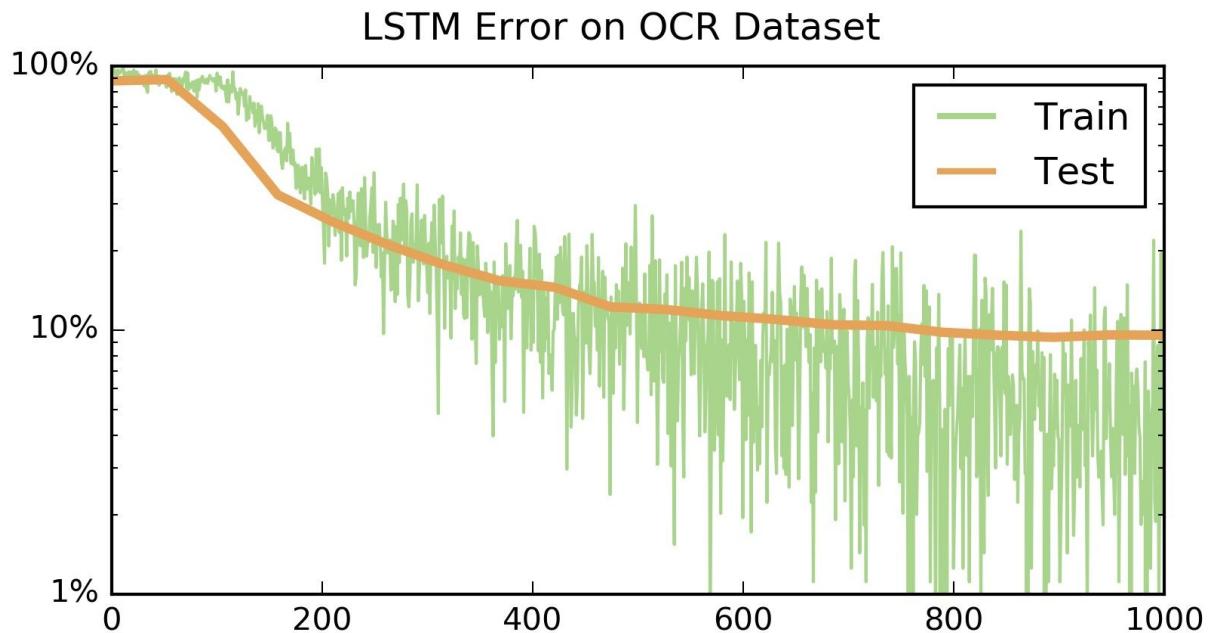
params = AttrDict(
    rnn_cell=tf.nn.rnn_cell.GRUCell,
    rnn_hidden=300,
    optimizer=tf.train.RMSPropOptimizer(0.002),
    gradient_clipping=5,
    batch_size=10,
    epochs=20,
    epoch_size=50,
)

def get_dataset():
    dataset = OcrDataset('~/dataset/book/ocr')
    # Flatten images into vectors.
    dataset.data = dataset.data.reshape(dataset.data.shape[:2] + (-1,))
    # One-hot encode targets.
    target = np.zeros(dataset.target.shape + (26,))
    for index, letter in np.ndenumerate(dataset.target):
        if letter:
            target[index][ord(letter) - ord('a')] = 1
    dataset.target = target
    # Shuffle order of examples.
    order = np.random.permutation(len(dataset.data))
    dataset.data = dataset.data[order]
    dataset.target = dataset.target[order]
    return dataset

# Split into training and test data.
dataset = get_dataset()
split = int(0.66 * len(dataset.data))
train_data, test_data = dataset.data[:split], dataset.data[split:]
train_target, test_target = dataset.target[:split], dataset.target[split:]

# Compute graph.
_, length, image_size = train_data.shape
num_classes = train_target.shape[2]
data = tf.placeholder(tf.float32, [None, length, image_size])
target = tf.placeholder(tf.float32, [None, length, num_classes])
model = SequenceLabellingModel(data, target, params)
```

After training of 1000 words our model classifies about 9% of all letter in the test set correctly. That's not too bad, but there is also room for improvement here.



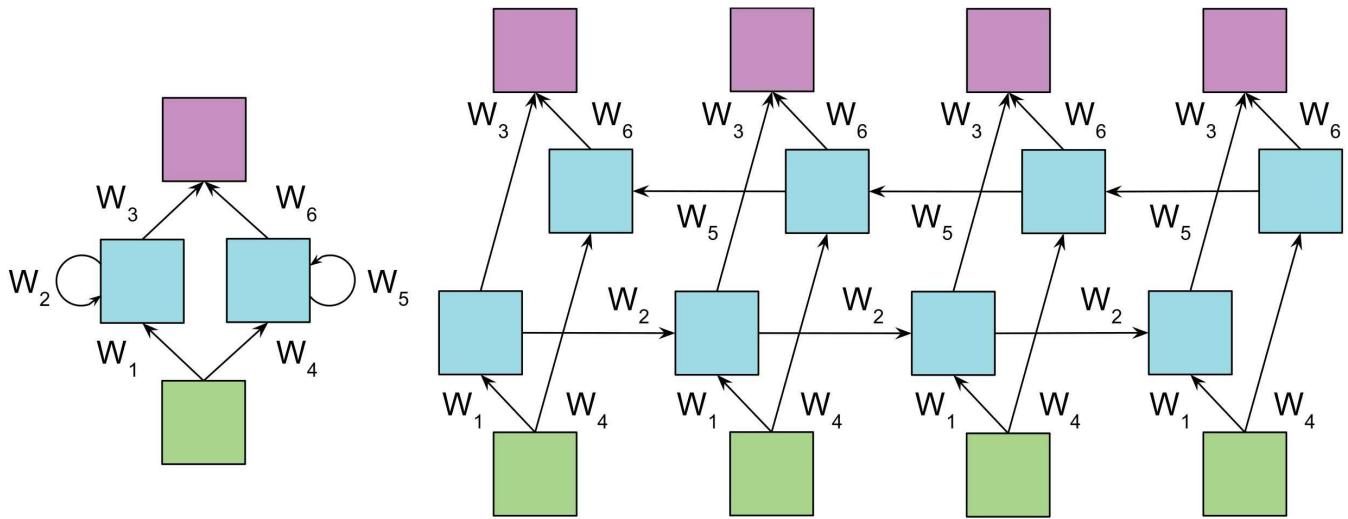
Our current model very similar to the model for sequence classification. This was by intent so that you see the differences needed to apply in order to adapt existing models to new tasks. What worked on another problem is more likely to work well on a new problem than if you would make a wild guess. However, we can do better! In the next section, we will try and improve on our results using a more advanced recurrent architecture.

Bidirectional RNNs

How can we improve the results on the OCR dataset that we got with the RNN plus Softmax architecture? Well, let's take a look at our motivation to use RNNs. The reason we chose them for the OCR dataset was that there are dependencies, or mutual information, between the letters within one word. The RNN stores information about all the previous inputs of the same word in its hidden activation.

If you think about it, the recurrency in our model doesn't help much for classifying the first few letters because the network hasn't had many inputs yet to infer additional information from. In sequence classification, this wasn't a problem since the network sees all frames before making a decision. In sequence labelling, we can address this shortcoming using *bidirectional RNNs*, a technique that holds state of the art in several classification problems.

The idea of bidirectional RNNs is simple. There are two RNNs that take a look at the input sequence, one going from the left reading the word in normal order, and one going from the right reading the letters in reverse order. At each time step, we now got two output activations that we concatenate before passing them up into the shared softmax layer. Using this architecture, the classifier can access information of the whole word at each letter.



Bi-Directional Recurrent Network

How can we implement bidirectional RNNs in TensorFlow? There is actually an implementation available with `tf.model.rnn.bidirectional_rnn`. However, we want to learn how to build complex models ourselves and so let's build our own implementation. I'll walk you through the steps. First, we split the prediction property into two functions so we can focus on smaller parts at the time.

```

@lazy_property
def prediction(self):
    output = self._bidirectional_rnn(self.data, self.length)
    num_classes = int(self.target.get_shape()[2])
    prediction = self._shared_softmax(output, num_classes)
    return prediction

def _bidirectional_rnn(self, data, length):
    pass

def _shared_softmax(self, data, out_size):
    pass

```

The `_shared_softmax()` function above is easy; we already had the code in the prediction property before. The difference is that this time, we infer the input size from the `data` tensor that gets passed into the function. This way, we can reuse the function for other architectures if needed. Then we use the same flattening trick to share the same softmax layer accross all time steps.

```

def _shared_softmax(self, data, out_size):
    max_length = int(data.get_shape()[1])
    in_size = int(data.get_shape()[2])
    weight = tf.Variable(tf.truncated_normal(
        [in_size, out_size], stddev=0.01))
    bias = tf.Variable(tf.constant(0.1, shape=[out_size]))
    # Flatten to apply same weights to all time steps.
    flat = tf.reshape(data, [-1, in_size])
    output = tf.nn.softmax(tf.matmul(flat, weight) + bias)
    output = tf.reshape(output, [-1, max_length, out_size])
    return output

```

Here comes the interesting part, the implementation of bidirectional RNNs. As you can see, we have create two RNNs using `tf.nn.dynamic_rnn`. The forward network should look familiar while the backward network is new.

Instead of just feeding in the data into the backward RNN, we first reverse the

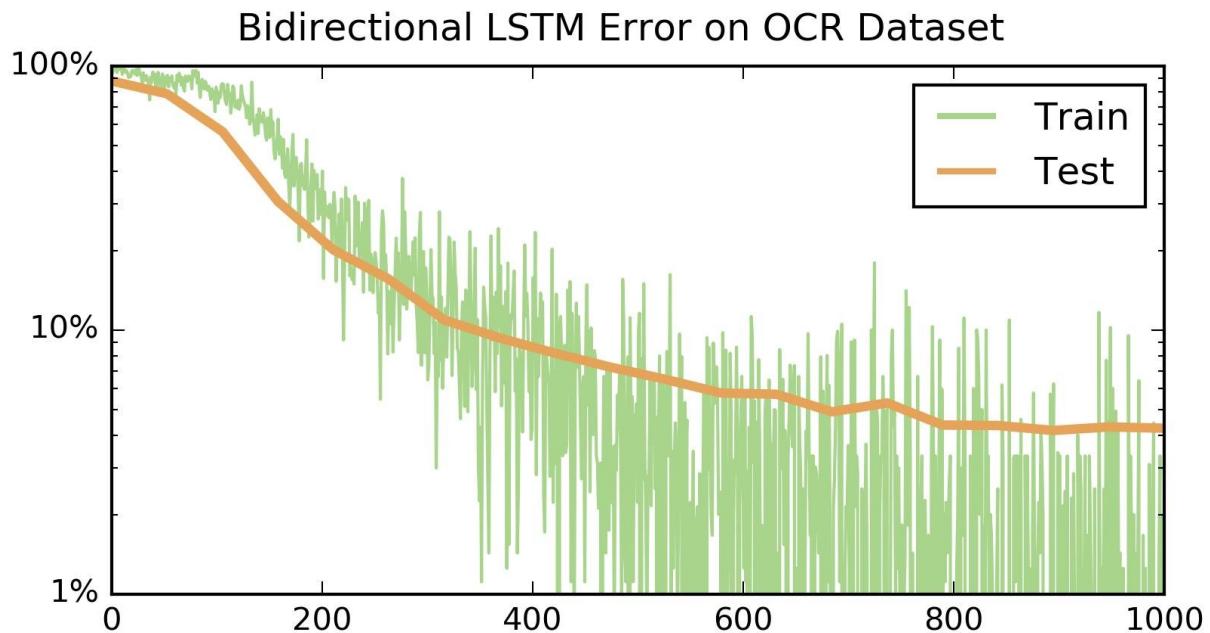
sequences. This is easier than implementing a new RNN operation that would go backwards. TensorFlow helps us with the `tf.reverse_sequence()` functions that takes care of only reversing the used frames up to `sequence_length`. Note that at the moment of writing this, the function expects the lengths to be a 64-bit integer tensor. It's likely that it will also work with 32-bit tensors and you can just pass in `self.length`.

```
def _bidirectional_rnn(self, data, length):
    length_64 = tf.cast(length, tf.int64)
    forward, _ = tf.nn.dynamic_rnn(
        cell=self.params.rnn_cell(self.params.rnn_hidden),
        inputs=data,
        dtype=tf.float32,
        sequence_length=length,
        scope='rnn-forward')
    backward, _ = tf.nn.dynamic_rnn(
        cell=self.params.rnn_cell(self.params.rnn_hidden),
        inputs=tf.reverse_sequence(data, length_64, seq_dim=1),
        dtype=tf.float32,
        sequence_length=self.length,
        scope='rnn-backward')
    backward = tf.reverse_sequence(backward, length_64, seq_dim=1)
    output = tf.concat(2, [forward, backward])
    return output
```

We also use the `scope` parameter this time. Why do we need this? As explained in the *TensorFlow Fundamentals* chapter, nodes in the compute graph have names. `scope` is the name of the variable scope used by `rnn.dynamic_cell` and it defaults to `RNN`. This time we have two RNNs that have different parameters so they have to live in different scopes.

After feeding the reversed sequence into the backward RNN, we again reverse the network outputs to align with the forward outputs. Then we concatenate both tensors along the dimension of the output neurons of the RNNs and return this. For example, with a batch size of 50, 300 hidden units per RNN and words of up to 14 letters, the resulting tensor would have the shape `50 x 14 x 600`.

Okay cool, we built our first architecture that is composed of multiple RNNs! Let's see how it performs using the training code from the last section. As you can see from comparing the graphs, the bidirectional model performs significantly better. After seeing 1000 words, it only misclassifies 4% of the letters in the test split.



To summarize, in this section we learned how to use RNNs for sequence labelling and the differences to the sequence classification setting. Namely, we want a classifier that takes the RNN outputs and is shared across all time steps.

This architecture can be improved drastically by adding a second RNN that visits the sequence from back to front and combining the outputs at each time step. This is because now information of the whole sequence are available for the classification of each letter.

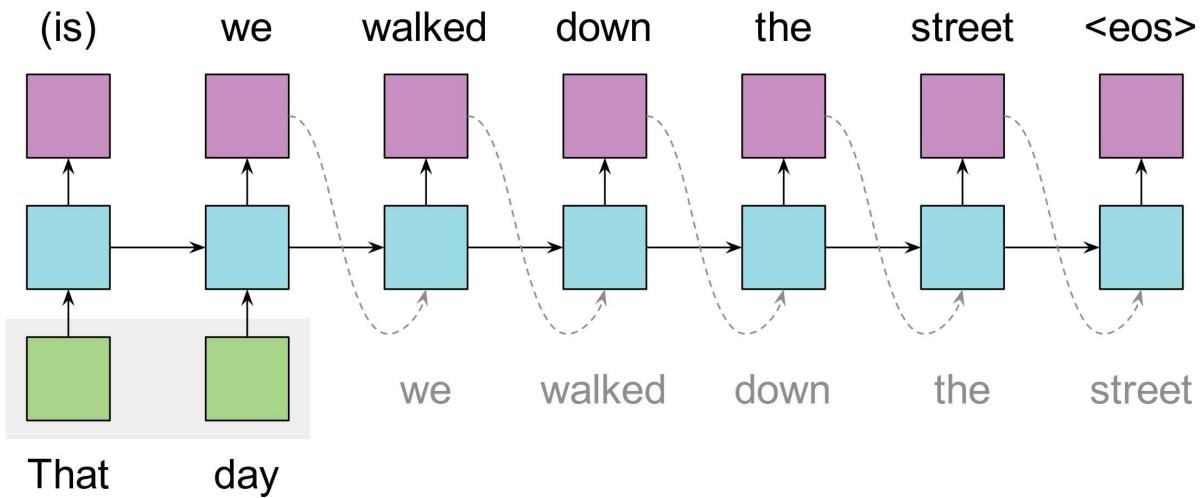
In the next section, we will take a look at training an RNN in an unsupervised fashion in order to learn language.

Predictive coding

We already learned how to use RNNs to classify the sentiment of movie reviews, and to recognize hand-written words. These applications have been supervised, meaning that we needed a labelled dataset. Another interesting learning setting is called *predictive coding*. We just show the RNN a lot of sequences and train it to predict the next frame of the sequence.

Let's take text as an example, where predicting the likelihood of the next word in a sentence is called *language modelling*. Why would it be useful to predict the next word in a sentence? One group of applications is recognizing language. Let's say you want to build a handwriting recognizer that translates scans of handwriting to typed text. While you can try to recover all the words from the input scans only, knowing the distribution of likely next words already narrows down the candidate words to decide between. It's the difference between dumb recognition of shapes and reading, basically.

Besides improving performance in tasks involving natural language, we can also sample from the distribution of what the network thinks should follow next in order to generate text. After training, we can start feeding a seed word into the RNN and look at the next word prediction. Then we feed the most likely word back into the RNN as the next input so see what it thinks should follow now. Doing this repeatedly, we can generate new content looking similar to the training data.



Seeded Sampling From a Recurrent Language Model

The interesting thing is that predictive coding trains the network to compress all the important information of any sequence. The next words in a sentence usually depends on the previous words, their order and relations between each other. A network that is able to accurately predict the next character in natural language thus needs to capture the rules of grammar and language well.

Character-level language modelling

We will now build a predictive coding language model using an RNN. Instead of the traditional approach to operate on words though, we will have our RNN operate on *individual characters*. So instead of word embeddings as inputs, we have a little over 26 one-hot encoded characters to represent the alphabet and some punctuation and whitespace.

It's known yet whether word-level or character-level language modelling is the better approach. The beauty of the character approach is that the network does not only learn how to combine words, but also how to spell them. In addition, the input to our network is lower-dimensional than if we would use word embeddings of size 300 or even one-hot encoded words. As a bonus, we don't have to account for unknown words anymore, because they are composed of letters that the network already knows about. This, in theory, even allows the network could invent new words.

Andrew Karpathy experimented with RNNs operating on characters in 2015 and was able to generate surprisingly nice samples of Shakespeare scripts, Linux kernel and driver code, and Wikipedia articles including correct markup syntax. The project is available on Github under <https://github.com/karpathy/char-rnn>. We will now train a similar model on the abstracts of machine learning publications and see if we can generate some more-or-less plausible new abstracts!

ArXiv abstracts API

ArXiv.org is an online library hosting many research papers from computer science, maths, physics and biology. You probably already heard of it if you are following machine learning research. Fortunately, the platform provides a web-based API to retrieve publications. Let's write a class that fetches the abstracts from ArXiv for a given search query.

```
import requests
import os
from bs4 import BeautifulSoup

class ArxivAbstracts:

    def __init__(self, cache_dir, categories, keywords, amount=None):
        pass

    def _fetch_all(self, amount):
        pass

    def _fetch_page(self, amount, offset):
        pass

    def _fetch_count(self):
        pass

    def _build_url(self, amount, offset):
        pass
```

In the constructor, we first check if there is already a previous dump of abstracts available. If it is, we will use that instead of hitting the API again. You could imagine more complicated logic to check if the existing file matches the new categories and keywords, but for now it is sufficient to delete or move the old dump manually to perform a new query. If no dump is available, we call the `_fetch_all()` method and write the lines it yields to disk.

```
def __init__(self, cache_dir, categories, keywords, amount=None):
    self.categories = categories
    self.keywords = keywords
    cache_dir = os.path.expanduser(cache_dir)
    ensure_directory(cache_dir)
    filename = os.path.join(cache_dir, 'abstracts.txt')
    if not os.path.isfile(filename):
        with open(filename, 'w') as file_:
            for abstract in self._fetch_all(amount):
                file_.write(abstract + '\n')
    with open(filename) as file_:
        self.data = file_.readlines()
```

Since we are interested in machine learning papers, we will search within the categories *Machine Learning*, *Neural and Evolutionary Computing*, and *Optimization and Control*. We further restrict the results to those containing any of the words *neural*, *network* or *deep* in the metadata. This gives us about 7 MB of text which is a fair amount of data to learn a simple RNN language model. It would be reasonable to use more data and get better results, but we don't want to wait for too many hours of training to pass before seeing some results. Feel free to use a broader search query and train this model on more data though.

```
ENDPOINT = 'http://export.arxiv.org/api/query'

def _build_url(self, amount, offset):
    categories = ' OR '.join('cat:' + x for x in self.categories)
```

```

keywords = ' OR '.join('all:' + x for x in self.keywords)
url = type(self).ENDPOINT
url += '?search_query=({}) AND ({}))'.format(categories, keywords)
url += '&max_results={}&offset={}'.format(amount, offset)
return url

def _fetch_count(self):
    url = self._build_url(0, 0)
    response = requests.get(url)
    soup = BeautifulSoup(response.text, 'lxml')
    count = int(soup.find('opensearch:totalresults').string)
    print(count, 'papers found')
    return count

```

The `_fetch_all()` method basically performs *pagination*. The API only gives us a certain amount of abstracts per request and we can specify an offset to get results of the second, third, etc “page”. As you can see, we can specify the page size which gets passed into the next function, `_fetch_page()`. In theory, we could set the page size to a huge number and try to get all results at once. In practice however, this makes the request very slow. Fetching in pages is also more fault tolerant and most importantly, does not stress the Arxiv API too much.

```

PAGE_SIZE = 100

def _fetch_all(self, amount):
    page_size = type(self).PAGE_SIZE
    count = self._fetch_count()
    if amount:
        count = min(count, amount)
    for offset in range(0, count, page_size):
        print('Fetch papers {} / {}'.format(offset + page_size, count))
        yield from self._fetch_page(page_size, count)

```

Here we perform the actual fetching. The result comes in XML and we use the popular and powerful BeautifulSoup library to extract the abstracts. If you haven’t installed it already, you can issue a `sudo -H pip3 install beautifulsoup4`. BeautifulSoup parses the XML result for us so that we can easily iterate over the tags that are of our interest. First, we look for `<entry>` tags corresponding to publications and within each of them, we read our the `<summary>` tag containing the abstract text.

```

def _fetch_page(self, amount, offset):
    url = self._build_url(amount, offset)
    response = requests.get(url)
    soup = BeautifulSoup(response.text)
    for entry in soup.findAll('entry'):
        text = entry.find('summary').text
        text = text.strip().replace('\n', ' ')
        yield text

```

Preprocessing the data

```
import random
import numpy as np

class Preprocessing:

    VOCABULARY = \
        "$%'()-.@123456789:;=?ABCDEFGHIJKLMNOPQRSTUVWXYZ" \
        "\\"^_abcdefghijklmnopqrstuvwxyz{|}"

    def __init__(self, texts, length, batch_size):
        self.texts = texts
        self.length = length
        self.batch_size = batch_size
        self.lookup = {x: i for i, x in enumerate(self.VOCABULARY)}

    def __call__(self, texts):
        batch = np.zeros((len(texts), self.length, len(self.VOCABULARY)))
        for index, text in enumerate(texts):
            text = [x for x in text if x in self.lookup]
            assert 2 <= len(text) <= self.length
            for offset, character in enumerate(text):
                code = self.lookup[character]
                batch[index, offset, code] = 1
        return batch

    def __iter__(self):
        windows = []
        for text in self.texts:
            for i in range(0, len(text) - self.length + 1, self.length // 2):
                windows.append(text[i: i + self.length])
        assert all(len(x) == len(windows[0]) for x in windows)
        while True:
            random.shuffle(windows)
            for i in range(0, len(windows), self.batch_size):
                batch = windows[i: i + self.batch_size]
                yield self(batch)
```

Predictive coding model

By now, you already know the procedure: We have defined our task, have written a parser to obtain a dataset and now we will implement the neural network model in TensorFlow. Because for predictive coding we try and predict the next character of the input sequence, there is only one input to the model, which is the `sequence` parameter in the constructor.

Moreover, the constructor takes a parameter object to change options in a central place and make our experiments reproducible. The third parameter `initial=None` is the initial inner activation of the recurrent layer. While we want to TensorFlow to initialize the hidden state to zero tensors for us, it will become handy to define it when we will sample from the learned language model later.

```
import tensorflow as tf
from utility import lazy_property

class PredictiveCodingModel:

    def __init__(self, params, sequence, initial=None):
        self.params = params
        self.sequence = sequence
        self.initial = initial
        self.prediction
        self.state
        self.cost
        self.error
        self.logprob
        self.optimize

    @lazy_property
    def data(self):
        pass

    @lazy_property
    def target(self):
        pass

    @lazy_property
    def mask(self):
        pass

    @lazy_property
    def length(self):
        pass

    @lazy_property
    def prediction(self):
        pass

    @lazy_property
    def state(self):
        pass

    @lazy_property
    def forward(self):
        pass

    @lazy_property
    def cost(self):
        pass

    @lazy_property
    def error(self):
        pass

    @lazy_property
```

```

def logprob(self):
    pass

@lazy_property
def optimize(self):
    pass

def _average(self, data):
    pass

```

In the code example above, you can see an overview of the functions that our model will implement. Don't worry it that looks overwhelming at first: We just want to expose some more values of our model than we did in the previous chapters.

Let's start with the data processing. As we said, the model just takes a one block of sequences as input. First, we use that to construct input data and target sequences from it. This is where we introduce a temporal difference because at timestep t , the model should have character s_t as input but s_{t+1} as target. An easy way to obtain `data` or `target` is to slice the provided sequence and cut away the last or the first frame, respectively.

We do this slicing using `tf.slice()` which takes the sequence to slice, a tuple of start indices for each dimension, and a tuple of sizes for each dimension. For the sizes `-1` means to keep all elements from the start index in that dimension until the end. Since we want to slice frames, we only care about the second dimension.

```

@lazy_property
def data(self):
    max_length = int(self.sequence.get_shape()[1])
    return tf.slice(self.sequence, (0, 0, 0), (-1, max_length - 1, -1))

@lazy_property
def target(self):
    return tf.slice(self.sequence, (0, 1, 0), (-1, -1, -1))

@lazy_property
def mask(self):
    return tf.reduce_max(tf.abs(self.target), reduction_indices=2)

@lazy_property
def length(self):
    return tf.reduce_sum(self.mask, reduction_indices=1)

```

We also define two properties on the target sequence as we already discussed in earlier sections: `mask` is a tensor of size `batch_size x max_length` where elements are zero or one depending on whether the respective frame is used or a padding frame. The `length` property sums up the mask along the time axis in order to obtain the length of each sequence.

Note that the mask and length properties are also valid for the data sequence since conceptually, they are of the same length as the target sequence. However, we couldn't compute them on the data sequence since it still contains the last frame that is not needed since there is no next character to predict. You are right, we sliced away the last frame of the data tensor, but that didn't contain the actual last frame of most sequences but mainly padding frames. This is the reason why we will use `mask` below to mask our cost function.

Now we will define the actual network that consists of a recurrent network and a shared softmax layer, just like we used for sequence labelling in the previous section. We don't show the code for the shared softmax layer here again but you can find it in the previous

section.

```
@lazy_property
def prediction(self):
    prediction, _ = self.forward
    return prediction

@lazy_property
def state(self):
    _, state = self.forward
    return state

@lazy_property
def forward(self):
    cell = self.params.rnn_cell(self.params.rnn_hidden)
    cell = tf.nn.rnn_cell.MultiRNNCell([cell] * self.params.rnn_layers)
    hidden, state = tf.nn.dynamic_rnn(
        inputs=self.data,
        cell=cell,
        dtype=tf.float32,
        initial_state=self.initial,
        sequence_length=self.length)
    vocabulary_size = int(self.target.get_shape()[2])
    prediction = self._shared_softmax(hidden, vocabulary_size)
    return prediction, state
```

The new part about the neural network code above is that we want to get both the prediction and the last recurrent activation. Previously, we only returned the prediction but the last activation allows us to generate sequences more effectively later. Since we only want to construct the graph for the recurrent network once, there is a `forward` property that return the tuple of both tensors and `prediction` and `state` are just there to provide easy access from the outside.

The next part of our model is the cost and evaluation functions. At each time step, the model predicts the next character out of the vocabulary. This is a classification problem and we use the cross entropy cost, accordingly. We can easily compute the error rate of character predictions as well.

The `logprob` property is new. It describes the probability that our model assigned to the correct next character in logarithmic space. This is basically the negative cross entropy transformed into logarithmic space and averaged there. Converting the result back into linear space yields the so-called *perplexity*, a common measure to evaluate the performance of language models.

The perplexity is defined as $2^{\frac{1}{n} \sum_{i=1}^n \log p(y_i)}$. Intuitively, it represents the number of options the model had to guess between at each time step. A perfect model has a perplexity of 1 while a model that always outputs the same probability for each of the n classes has a perplexity of n . The perplexity can even become infinity when the model assigns a zero probability to the next character once. We prevent this extreme case by clamping the prediction probabilities within a very small positive number and one.

```
@lazy_property
def cost(self):
    prediction = tf.clip_by_value(self.prediction, 1e-10, 1.0)
    cost = self.target * tf.log(prediction)
    cost = -tf.reduce_sum(cost, reduction_indices=2)
    return self._average(cost)

@lazy_property
def error(self):
```

```

error = tf.not_equal(
    tf.argmax(self.prediction, 2), tf.argmax(self.target, 2))
error = tf.cast(error, tf.float32)
return self._average(error)

@lazy_property
def logprob(self):
    logprob = tf.mul(self.prediction, self.target)
    logprob = tf.reduce_max(logprob, reduction_indices=2)
    logprob = tf.log(tf.clip_by_value(logprob, 1e-10, 1.0)) / tf.log(2.0)
    return self._average(logprob)

def _average(self, data):
    data *= self.mask
    length = tf.reduce_sum(self.length, 1)
    data = tf.reduce_sum(data, reduction_indices=1) / length
    data = tf.reduce_mean(data)
    return data

```

All the three properties above are averaged over the frames of all sequences. With fixed-length sequences, this would be a single `tf.reduce_mean()`, but as we work with variable-length sequences, we have to be a bit more careful. First, we mask out padding frames by multiplying with the mask. Then we aggregate along the frame size. Because the three functions above all multiply with the target, each frame has just one element set and we use `tf.reduce_sum()` to aggregate each frame into a scalar.

Next, we want to average along the frames of each sequence using the actual sequence length. To protect against division by zero in case of empty sequences, we use the maximum of each sequence length and one. Finally, we can use `tf.reduce_mean()` to average over the examples in the batch.

We will directly head to training this model. Note that we did not define the `optimize` operation. It is identical to those used for sequence classification or sequence labelling earlier in the chapter.

Training the model

Before sampling from our language model, we have to put together the blocks we just built: The dataset, the preprocessing step and the neural model. Let's write a class for that that puts together these steps, prints the newly introduced perplexity measure and regularly stores training progress. This checkpointing is useful to continue training later but also to load the trained model for sampling, which we will do shortly.

```
import os

class Training:

    @overwrite_graph
    def __init__(self, params):
        self.params = params
        self.texts = ArxivAbstracts('/home/user/dataset/arxiv')()
        self.prep = Preprocessing(
            self.texts, self.params.max_length, self.params.batch_size)
        self.sequence = tf.placeholder(
            tf.float32,
            [None, self.params.max_length, len(self.prep.VOCABULARY)])
        self.model = PredictiveCodingModel(self.params, self.sequence)
        self._init_or_load_session()

    def __call__(self):
        print('Start training')
        self.logprobs = []
        batches = iter(self.prep)
        for epoch in range(self.epoch, self.params.epochs + 1):
            self.epoch = epoch
            for _ in range(self.params.epoch_size):
                self._optimization(next(batches))
            self._evaluation()
        return np.array(self.logprobs)

    def _optimization(self, batch):
        logprob, _ = self.sess.run(
            (self.model.logprob, self.model.optimize),
            {self.sequence: batch})
        if np.isnan(logprob):
            raise Exception('training diverged')
        self.logprobs.append(logprob)

    def _evaluation(self):
        self.saver.save(self.sess, os.path.join(
            self.params.checkpoint_dir, 'model'), self.epoch)
        self.saver.save(self.sess, os.path.join(
            self.params.checkpoint_dir, 'model'), self.epoch)
        perplexity = 2 ** -(sum(self.logprobs[-self.params.epoch_size:]) /
                           self.params.epoch_size)
        print('Epoch {:2d} perplexity {:.5f}'.format(self.epoch, perplexity))

    def _init_or_load_session(self):
        self.sess = tf.Session()
        self.saver = tf.train.Saver()
        checkpoint = tf.train.get_checkpoint_state(self.params.checkpoint_dir)
        if checkpoint and checkpoint.model_checkpoint_path:
            path = checkpoint.model_checkpoint_path
            print('Load checkpoint', path)
            self.saver.restore(self.sess, path)
            self.epoch = int(re.search(r'-(\d+)$', path).group(1)) + 1
        else:
            ensure_directory(self.params.checkpoint_dir)
            print('Randomly initialize variables')
            self.sess.run(tf.initialize_all_variables())
            self.epoch = 1
```

The constructor, `__call__()`, `_optimization()` and `_evaluation()` should be easy to understand. We load the dataset and define inputs to the compute graph, train on the preprocessed

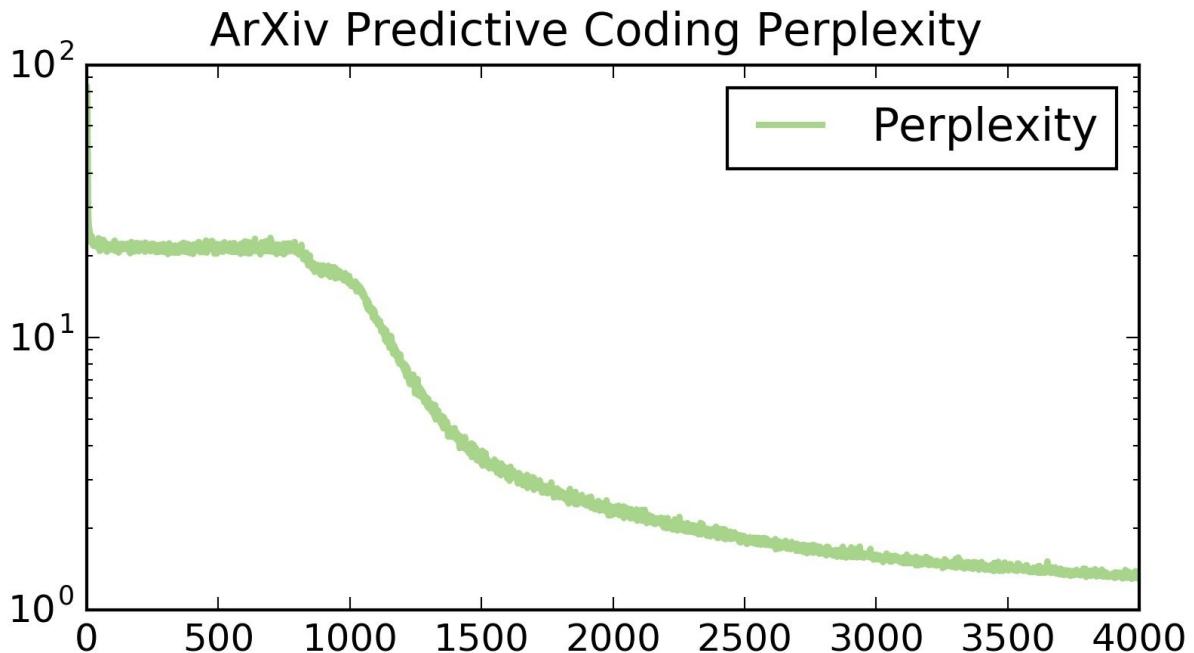
dataset and keep track of the logarithmic probabilities. We use those at evaluation time between each training epoch to compute and print the perplexity.

In `_init_or_load_session()` we introduce a `tf.train.Saver()` that stores the current values of all `tf.Variable()` in the graph to a checkpoint file. While the actual checkpointing is done in `_evaluation()`, here we create the class and look for existing checkpoints to load. The `tf.train.get_checkpoint_state()` looks for TensorFlow's meta data file in our checkpoint directory. As of writing, it only contains the file of the least recent checkpoint file.

Checkpoint files are prepended by a number that we can specify, in our case the epoch number. When loading a checkpoint, we apply a regular expression with Python's `re` package to extract that epoch number. With the checkpointing logic set up, we can start training. Here is the configuration:

```
def get_params():
    checkpoint_dir = '/home/user/model/arxiv-predictive-coding'
    max_length = 50
    sampling_temperature = 0.7
    rnn_cell = GRUCell
    rnn_hidden = 200
    rnn_layers = 2
    learning_rate = 0.002
    optimizer = tf.train.AdamOptimizer
    gradient_clipping = 5
    batch_size = 100
    epochs = 20
    epoch_size = 200
    return AttrDict(**locals())
```

To run the code, you can just call `Training(get_params())()`. On my notebook, it takes about one hour for the 20 epochs. During this training, the model saw $20 \text{ epochs} * 200 \text{ batches} * 100 \text{ examples} * 50 \text{ characters} = 20\text{M characters}$.



As you can see on the graph, the model converges at a perplexity of about 1.5 per character. This means that with our model, we could compress a text at an average of 1.5 bits per character.

For comparison with word-level language models, we would have to average by the

number of words rather than the number of characters. As a rough estimate, we can multiply it by the average number of characters per word, which is ... on our test set.

Generating similar sequences

After all the work, we can now use the trained model to sample new sequences. We will write a small class that work similar to our `Training` class in that it loads the latest model checkpoint from disk and defines placeholders to feed data into the compute graph. Of course, this time we don't train the model but use it to generate new data.

```
class Sampling:
    @overwrite_graph
    def __init__(self, params):
        pass

    def __call__(self, seed, length):
        pass

    def _sample(self, dist):
        pass
```

In the constructor, we create an instance of our preprocessing class that we will use convert the current generated sequence into a Numpy vector to feed into the graph. The `sequence` placeholder for this is only has one sequence per batch because we don't want to generate multiple sequences at the same time.

One thing to explain is the sequence length of two. Remember that the model use all but the last characters as input data and all but the first characters as targets. We feed in the last character of the current text and any second character as sequence. The network will predict the target for the first character. The second character is used as target but since we don't train anything, it will be ignored.

You may wonder how we can get along with only passing the last character of the current text into the network. The trick here is that we will get the last activation of the recurrent network and use that to initialize the state in the next run. For this, we make use of the initial state argument of our model. For the `GRUCell` that we used, the state is a vector of size `rnn_layers * rnn_units`.

```
@overwrite_graph
def __init__(self, params, length):
    self.params = params
    self.prep = Preprocessing([], 2, self.params.batch_size)
    self.sequence = tf.placeholder(
        tf.float32, [1, 2, len(self.prep.VOCABULARY)])
    self.state = tf.placeholder(
        tf.float32, [1, self.params.rnn_hidden * self.params.rnn_layers])
    self.model = PredictiveCodingModel(
        self.params, self.sequence, self.state)
    self.sess = tf.Session()
    checkpoint = tf.train.get_checkpoint_state(self.params.checkpoint_dir)
    if checkpoint and checkpoint.model_checkpoint_path:
        tf.train.Saver().restore(
            self.sess, checkpoint.model_checkpoint_path)
    else:
        print('Sampling from untrained model.')
        print('Sampling temperature', self.params.sampling_temperature)
```

The `__call__()` functions defines the logic for sampling a text sequence. We start with the seed and predict one character at a time, always feeding the current text into the network. We use the same preprocessing class to convert the current texts into padded Numpy blocks and feed them into the network. Since we only have one sequence with a single

output frame in the batch, we only care at the prediction at index $[0, 0]$. We then sample from the softmax output using the `_sample()` function described next.

```
def __call__(self, seed, length=100):
    text = seed
    state = np.zeros((1, self.params.rnn_hidden * self.params.rnn_layers))
    for _ in range(length):
        feed = {self.state: state}
        feed[self.sequence] = self.prep([text[-1] + '?'])
        prediction, state = self.sess.run(
            [self.model.prediction, self.model.state], feed)
        text += self._sample(prediction[0, 0])
    return text
```

How do we sample from the network output? Earlier we said we can generate sequences by taking their best bet and feeding that in as the next frame. Actually, we don't just choose the most likely next frame but randomly sample one from the probability distribution that the RNN outputs. This way, words with a high output probability are more likely to be chosen but less likely words are still possible. This results in more dynamic generated sequences. Otherwise, we might just generate the same average sentence again and again.

There is a simple mechanism to manually control how advantageous the generation process should be. For example, if we always choose the next word randomly (and ignore the network output completely), we get very new and unique sentences, but they would not make any sense. If we always choose the network's highest output as the next word, we get a lot of common, but meaningless words like "the," "a," etc.

The way we can control this behavior is by introducing a *temperature* parameter T . We use this parameter to make the predictions of the output distribution at the softmax layer more similar or more radical. This will result in more interesting but random sequences on the one side of the spectrum, and to more plausible but boring sequences on the other side. The way it works is that we scale the outputs in linear space, then transform them back into exponential space and normalize again:

$$p(x_i) = \frac{e^{\frac{x_i}{T}}}{\sum_j e^{\frac{x_j}{T}}}$$

Since the network already outputs a softmax distribution, we undo it by applying the natural logarithm. We don't have to undo the normalization since we will normalize our results again, anyways. Then we divide each value by the chosen temperature value and re-apply the softmax function.

```
def _sample(self, dist):
    dist = np.log(dist) / self.params.sampling_temperature
    dist = np.exp(dist) / np.exp(dist).sum()
    choice = np.random.choice(len(dist), p=dist)
    choice = self.prep.VOCABULARY[choice]
    return choice
```

Let's run the code by calling `sampling(get_params())('We', 500)` for the network to generate a new abstract. While you can certainly tell that this text is not written by a human, it is quite remarkable what the network learns from examples.

We study nonconvex encoder in the networks (RFNs) hasding configurations with non-convex large-layers of images, each directions literatic for layers. More recent results competitive strategy, in which data at training and more difficult to parallelize. Recent Newutic systems, the desirmally parametrically in the DNNs improves optimization technique, we extend their important and subset of theidesteding and dast and scale in recent advances in sparse recovery to complicated patterns of the \$L_p\$

We did not tell the RNN what a space is, but it captured statistically dependencies in the data to place whitespace accordingly in the generated text. Even between some non-existent words that the network dreamed up, the whitespace looks reasonable. Moreover, those words are composed of valid combinations of vowels and consonants, another abstract feature learned from the example texts.

Conclusion

RNNs are powerful sequential models that are applicable to a wide range of problems and are responsible for state-of-the-art results. We learned how to optimize RNNs, what problems arise doing so, and how architectures like LSTM and GRU help to overcome them. Using these building blocks, we solved several problems in natural language processing and related domains including classifying the sentiment of movie reviews, recognizing hand-written words, and generating fake scientific abstracts.

In the next chapter we will put our trained models in production so they can be consumed by other applications.

Part IV. Additional Tips, Techniques, and Features

Chapter 7. Deploying Models in Production

So far we have seen how to work with Tensorflow for building and training models from basic machine learning to complex deep learning networks. In this chapter we are going to focus on putting our trained models in production so they can be consumed by other apps.

Our goal will be to create a simple webapp that will allow the user to upload an image and run the Inception model over it for classifying.

Setting up a Tensorflow serving development environment

Tensorflow serving is the tool for building servers that allow to use our models in production. There are two flavors to use it during development: manually installing all the dependencies and tools for building it from source, or using a Docker image. We are going to use the latter since it is easier, cleaner, and allows you to develop in other environments than Linux.

In case you don't know what a Docker image is, think of it as a lightweight version of a virtual machine image that runs without the overhead of running a full OS inside it. You should install Docker in your development machine first if you haven't. Follow instructions from <https://docs.docker.com/engine/installation/>.

To use the Docker image, we have available the

https://github.com/tensorflow/serving/blob/master/tensorflow_serving/tools/docker/Dockerfile, which is the configuration file to create the image locally, so to use it we should:

```
docker build --pull -t $USER/tensorflow-serving-devel https://raw.githubusercontent.com/tensorflow/s
```

Be aware that it may take a while to download all of the dependencies.

Now to run the container using the image to start working on it we use:

```
docker run -v $HOME:/mnt/home -p 9999:9999 -it $USER/tensorflow-serving-devel
```

That will load mount your home directory in the `/mnt/home` path of the container, and will let you working in a terminal inside of it. This is useful as you can work your code directly on your favorite IDE/editor, and just use the container for running the build tools. It will also leave the port 9999 open to access it from your host machine for later usage of the server we are going to build.

You can leave the container terminal with `exit`, which will stop it from running, and start it again as many times you want using command above.

Bazel workspace

Tensorflow Serving programs are coded in C++ and should be built using Google's Bazel build tool. We are going to run Bazel from inside the recently created container.

Bazel manages third party dependencies at code level, downloading and building them, as long as they are also built with Bazel. To define which third party dependencies our project would support, we must define a `WORKSPACE` file at the root of our project repository.

The dependencies we need are Tensorflow Serving repository, and for the case of our example, the Tensorflow Models repository includes the Inception model code.

Sadly, at the moment of this writing, Tensorflow Serving does not support being referenced directly thru Bazel as a Git repository, so we must include it as a Git submodule in our project:

```
# on your local machine
mkdir ~/serving_example
cd ~/serving_example
git init
git submodule add https://github.com/tensorflow/serving.git tf_serving
git submodule update --init --recursive
```

We now define the third party dependencies as locally stored files using the `local_repository` rule on the `WORKSPACE` file. We also have to initialize Tensorflow dependencies using the `tf_workspace` rule imported from the project:

```
# Bazel WORKSPACE file

workspace(name = "serving")

local_repository(
    name = "tf_serving",
    path = __workspace_dir__ + "/tf_serving",
)

local_repository(
    name = "org_tensorflow",
    path = __workspace_dir__ + "/tf_serving/tensorflow",
)

load('//tf_serving/tensorflow:workspace.bzl', 'tf_workspace')
tf_workspace("tf_serving/tensorflow/", "@org_tensorflow")

bind(
    name = "libssl",
    actual = "@boringssl_git//:ssl",
)

bind(
    name = "zlib",
    actual = "@zlib_archive//:zlib",
)

# only needed for inception model export
local_repository(
    name = "inception_model",
    path = __workspace_dir__ + "/tf_serving/tf_models/inception",
)
```

As a last step we have to run `./configure` for Tensorflow from within the container:

```
# on the docker container
cd /mnt/home/serving_example/tf_serving/tensorflow
./configure
```

Exporting trained models

Once our model is trained and we are happy with the evaluation, we will need to export its computing graph along its variables values in order to make it available for production usage.

The graph of the model should be slightly changed from its training version, as it must take its inputs from placeholders and run a single step of inference on them to compute the output. For the example of the Inception model, and for any image recognition model in general, we want the input to be a single string representing a JPEG encoded image, so we can easily send it from our consumer app. This is quite different from the training input that reads from a TFRecords file.

The general form for defining the inputs should look like:

```
def convert_external_inputs(external_x):
    # transform the external input to the input format required on inference

def inference(x):
    # from the original model...

external_x = tf.placeholder(tf.string)
x = convert_external_inputs(external_x)
y = inference(x)
```

In the code above we define the placeholder for the input. We call a function to convert the external input represented in the placeholder to the format required for the original model inference method. For example we will convert from the JPEG string to the image format required for Inception. Finally we call the original model inference method with the converted input.

For example, for the Inception model we should have methods like:

```
import tensorflow as tf
from tensorflow_serving.session_bundle import exporter
from inception import inception_model

def convert_external_inputs(external_x):
    # transform the external input to the input format required on inference
    # convert the image string to a pixels tensor with values in the range 0,1
    image = tf.image.convert_image_dtype(tf.image.decode_jpeg(external_x, channels=3), tf.float32)
    # resize the image to the model expected width and height
    images = tf.image.resize_bilinear(tf.expand_dims(image, 0), [299, 299])
    # Convert the pixels to the range -1,1 required by the model
    images = tf.mul(tf.sub(images, 0.5), 2)
    return images

def inference(images):
    logits, _ = inception_model.inference(images, 1001)
    return logits
```

In the code above we define the placeholder for the input. We call a function to convert the external input represented in the placeholder to the format required for the original model inference method. For example we will convert from the JPEG string to the image format required for Inception. Finally we call the original model inference method with the converted input.

The inference method requires values for its parameters. We will recover those from a training checkpoint. As you may recall from the basics chapter, we periodically save training checkpoint files of our model. Those contain the learned values of parameters at the time, so in case of disaster we don't lose the training progress.

When we declare the training complete, the last saved training checkpoint will contain the most updated model parameters, which are the ones we wish to put in production.

To restore the checkpoint, the code should be:

```
saver = tf.train.Saver()

with tf.Session() as sess:
    # Restore variables from training checkpoints.
    ckpt = tf.train.get_checkpoint_state(sys.argv[1])
    if ckpt and ckpt.model_checkpoint_path:
        saver.restore(sess, sys.argv[1] + "/" + ckpt.model_checkpoint_path)
    else:
        print("Checkpoint file not found")
        raise SystemExit
```

For the Inception model, you can download a pretrained checkpoint from

<http://download.tensorflow.org/models/image/imagenet/inception-v3-2016-03-01.tar.gz>

```
# on the docker container
cd /tmp
curl -O http://download.tensorflow.org/models/image/imagenet/inception-v3-2016-03-01.tar.gz
tar -xzf inception-v3-2016-03-01.tar.gz
```

Finally we export the model using the `tensorflow_serving.session_bundle.exporter.Exporter` class. We create an instance of it passing the `saver` instance. Then we have to create the signature of the model using the `exporter.classification_signature` method. The signature specifies which is the `input_tensor`, and which are the output tensors. The output is composed by the `classes_tensor`, which contains the list of output class names, and the `scores_tensor`, which contains the score/probability the model assigns to each class. Typically in a model with a high number of classes, you would configure those to return only classes selected with `tf.nn.top_k`. Those are the K classes with the highest assigned score by the model.

The last step is to apply the signature calling the `exporter.Exporter.init` method and run the export with the `export` method, which receives an output path, a version number for the model and the session.

```
scores, class_ids = tf.nn.top_k(y, NUM_CLASSES_TO_RETURN)

# for simplification we will just return the class ids, we should return the names instead
classes = tf.contrib.lookup.index_to_string(tf.to_int64(class_ids),
                                             mapping=tf.constant([str(i) for i in range(1001)]))

model_exporter = exporter.Exporter(saver)
signature = exporter.classification_signature(
    input_tensor=external_x, classes_tensor=classes, scores_tensor=scores)
model_exporter.init(default_graph_signature=signature, init_op=tf.initialize_all_tables())
model_exporter.export(sys.argv[1] + "/export", tf.constant(time.time()), sess)
```

Because of dependencies to auto-generated code in the `Exporter` class code, you will need to run our exporter using `bazel`, inside the Docker container.

To do so we will save our code as `export.py` inside the `bazel` workspace we started before.

Will need to a `BUILD` file with a rule for building it like:

```
# BUILD file
py_binary(
    name = "export",
    srcs = [
        "export.py",
    ],
    deps = [
        "//tensorflow_serving/session_bundle:exporter",
        "@org_tensorflow//tensorflow:tensorflow_py",
        # only needed for inception model export
        "@inception_model//inception",
    ],
)
```

We can then run the exporter from between the container with the command:

```
# on the docker container
cd /mnt/home/serving_example
bazel run :export /tmp/inception-v3
```

And it will create the export in `/tmp/inception-v3/{current_timestamp}/` based on the checkpoint that should be extracted in `/tmp/inception-v3`.

Note that the first time you run it will take some time, because it must compile Tensorflow.

Defining a server interface

The next step is to create a server for the exported model.

Tensorflow Serving is designed to work with gRPC, a binary RPC protocol that works over HTTP/2. It supports a variety of languages for creating servers and auto-generating client stubs. As Tensorflow works over C++, we will need to define our server in it. Luckily the server code will be short.

In order to use gRPC, we must define our service contract in a *protocol buffer*, which is the IDL and binary encoding used for gRPC. So lets define our service. As we mentioned in the exporting section, we want our service to have a method that inputs a JPEG encoded string of the image to classify and returns a list of inferred classes with their corresponding scores.

Such a service should be defined in a `classification_service.proto` file like:

```
syntax = "proto3";

message ClassificationRequest {
    // JPEG encoded string of the image.
    bytes input = 1;
}

message ClassificationResponse {
    repeated ClassificationClass classes = 1;
}

message ClassificationClass {
    string name = 1;
    float score = 2;
}

service ClassificationService {
    rpc classify(ClassificationRequest) returns (ClassificationResponse);
}
```

You can use this same interface definition for any kind of service that receives an image, or an audio fragment, or a piece of text.

For using an structured input like a database record, just change the `ClassificationRequest` message. For example, if we were trying to build the classification service for the Iris dataset:

```
message ClassificationRequest {
    float petalWidth = 1;
    float petalHeight = 2;
    float sepalWidth = 3;
    float sepalHeight = 4;
}
```

The proto file will be converted to the corresponding classes definitions for the client and the server by the proto compiler. To use the protobuf compiler, we have to add a new rule to the `BUILD` file like:

```
load("@protobuf//:protobuf.bzl", "cc_proto_library")

cc_proto_library(
    name="classification_service_proto",
    srcs=["classification_service.proto"],
    cc_libs = ["@protobuf//:protobuf"],
    protoc="@protobuf//:protoc",
```

```
  default_runtime="@protobuf//:protobuf",
  use_grpc_plugin=1
)
```

Notice the `load` at the top of the code fragment. It imports the `cc_proto_library` rule definition from the externally imported `protobuf` library. Then we use it for defining a build rule to our proto file. Let's run the build using `bazel build :classification_service_proto` and check the resulting `bazel-genfiles/classification_service.grpc.pb.h`:

```
...
class ClassificationService {
...
class Service : public ::grpc::Service {
public:
  Service();
  virtual ~Service();
  virtual ::grpc::Status classify(::grpc::ServerContext* context, const ::ClassificationRequest*
};
```

`ClassificationService::Service` is the interface we have to implement with the inference logic. We can also check `bazel-genfiles/classification_service.pb.h` for the definitions of the request and response messages:

```
...
class ClassificationRequest : public ::google::protobuf::Message {
...
  const ::std::string& input() const;
  void set_input(const ::std::string& value);
...
}
class ClassificationResponse : public ::google::protobuf::Message {
...
  const ::ClassificationClass& classes() const;
  void set_allocated_classes(::ClassificationClass* classes);
...
}
class ClassificationClass : public ::google::protobuf::Message {
...
  const ::std::string& name() const;
  void set_name(const ::std::string& value);
  float score() const;
  void set_score(float value);
...
}
```

You can see how the proto definition became a C++ class interface for each type. Their implementations are autogenerated too so we can just use them right away.

Implementing an inference server

To implement `ClassificationService::Service` we will need to load our model export and call inference on it. We do that by the means of a `SessionBundle`, an object that we create from the export and contains a TF session with the fully loaded graph, as well as the metadata including the classification signature defined on the export tool.

To create a `SessionBundle` from the exported file path, we can define a handy function that handles the boilerplate:

```
#include <iostream>
#include <memory>
#include <string>

#include <grpc++/grpc++.h>

#include "classification_service.grpc.pb.h"
#include "tensorflow_serving/servables/tensorflow/session_bundle_factory.h"

using namespace std;
using namespace tensorflow::serving;
using namespace grpc;

unique_ptr<SessionBundle> createSessionBundle(const string& pathToExportFiles) {
    SessionBundleConfig session_bundle_config = SessionBundleConfig();
    unique_ptr<SessionBundleFactory> bundle_factory;
    SessionBundleFactory::Create(session_bundle_config, &bundle_factory);

    unique_ptr<SessionBundle> sessionBundle;
    bundle_factory->CreateSessionBundle(pathToExportFiles, &sessionBundle);

    return sessionBundle;
}
```

In the code we use a `SessionBundleFactory` to create the `SessionBundle` configured to load the model exported in the path specified by `pathToExportFiles`. It returns a unique pointer to the instance of the created `SessionBundle`.

We now have to define the implementation of the service, `ClassificationServiceImpl` that will receive the `SessionBundle` as parameter to be used to do the inference.

```
class ClassificationServiceImpl final : public ClassificationService::Service {

private:
    unique_ptr<SessionBundle> sessionBundle;

public:
    ClassificationServiceImpl(unique_ptr<SessionBundle> sessionBundle) :
        sessionBundle(move(sessionBundle)) {}

    Status classify(ServerContext* context, const ClassificationRequest* request,
                    ClassificationResponse* response) override {

        // Load classification signature
        ClassificationSignature signature;
        const tensorflow::Status signatureStatus =
            GetClassificationSignature(sessionBundle->meta_graph_def, &signature);

        if (!signatureStatus.ok()) {
            return Status(StatusCode::INTERNAL, signatureStatus.error_message());
        }

        // Transform protobuf input to inference input tensor.
        tensorflow::Tensor input(tensorflow::DT_STRING, tensorflow::TensorShape());
        input.scalar<string>()() = request->input();
```

```

        vector<tensorflow::Tensor> outputs;

        // Run inference.
        const tensorflow::Status inferenceStatus = sessionBundle->session->Run(
            {{signature.input().tensor_name(), input}},
            {signature.classes().tensor_name(), signature.scores().tensor_name()},
            {},
            &outputs);

        if (!inferenceStatus.ok()) {
            return Status(StatusCode::INTERNAL, inferenceStatus.error_message());
        }

        // Transform inference output tensor to protobuf output.
        for (int i = 0; i < outputs[0].vec<string>().size(); ++i) {
            ClassificationClass *classificationClass = response->add_classes();
            classificationClass->set_name(outputs[0].flat<string>()(i));
            classificationClass->set_score(outputs[1].flat<float>()(i));
        }

        return Status::OK;
    }
};

The implementation of the classify method does four steps:

```

- Loads the `classificationSignature` stored in the model export meta by using the `GetClassificationSignature` function. The signature specifies the name of the input tensor where to set the received image, and the names of the output tensors in the graph where to obtain the inference results from.
- Copies the JPEG encoded image string from the `request` parameter into a tensor to be sent to inference.
- Runs inference. It obtains the TF session from `sessionBundle` and runs one step on it, passing references to the input and outputs tensors.
- Copies and formats the results from the output tensors to the `response` output param in the shape specified by the `classificationResponse` message.

The last piece of code is the boilerplate to setup a gRPC server and create an instance of our `ClassificationServiceImpl`, Configured with the `SessionBundle`.

```

int main(int argc, char** argv) {
    if (argc < 3) {
        cerr << "Usage: server <port> /path/to/export/files" << endl;
        return 1;
    }

    const string serverAddress(string("0.0.0.0:") + argv[1]);
    const string pathToExportFiles(argv[2]);

    unique_ptr<SessionBundle> sessionBundle = createSessionBundle(pathToExportFiles);

    ClassificationServiceImpl classificationServiceImpl(move(sessionBundle));

    ServerBuilder builder;
    builder.AddListeningPort(serverAddress, grpc::InsecureServerCredentials());
    builder.RegisterService(&classificationServiceImpl);

    unique_ptr<Server> server = builder.BuildAndStart();
    cout << "Server listening on " << serverAddress << endl;

    server->Wait();

    return 0;
}

```

To compile this code we have to define a rule in our `BUILD` file for it

```
cc_binary(  
  name = "server",  
  srcs = [  
    "server.cc",  
  ],  
  deps = [  
    ":classification_service_proto",  
    "@tf_serving//tensorflow_serving/servables/tensorflow:session_bundle_factory",  
    "@grpc//:grpc++",  
  ],  
)
```

With this code we can run the inference server from the container with `bazel run :server 9999 /tmp/inception-v3/export/{timestamp}`.

The client app

As gRPC works over HTTP/2, it may allow in the future to call gRPC based services directly from the browser. But until the mainstream of browsers support the required HTTP/2 features and Google releases a browser side Javascript gRPC client, accessing our inference service from a webapp should be done through a server side component.

We are going to do then a really simple Python web server based on `BaseHTTPServer` that will handle the image file upload and send for processing to inference, returning the inference result in plain text.

Our server will respond `GET` requests with a simple form for sending the image to classify. The code for it:

```
from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler

import cgi
import classification_service_pb2
from grpc.beta import implementations

class ClientApp(BaseHTTPRequestHandler):
    def do_GET(self):
        self.respond_form()

    def respond_form(self, response=""):

        form = """
<html><body>
<h1>Image classification service</h1>
<form enctype="multipart/form-data" method="post">
<div>Image: <input type="file" name="file" accept="image/jpeg"></div>
<div><input type="submit" value="Upload"></div>
</form>
%s
</body></html>
"""

        response = form % response

        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.send_header("Content-length", len(response))
        self.end_headers()
        self.wfile.write(response)
```

To call inference from our webapp server, we need the corresponding Python protocol buffer client for the `classificationService`. To generate it we will need to run the protocol buffer compiler for Python:

```
pip install grpcio cython grpcio-tools
python -m grpc.tools.protoc -I. --python_out=. --grpc_python_out=. classification_service.proto
```

It will generate the `classification_service_pb2.py` file that contains the `stub` for calling the service.

On `POST` the server will parse the sent form and create a `ClassificationRequest` with it. Then setup a `channel` to the classification server and submit the request to it. Finally, it will render the classification response as HTML and send it back to the user.

```
def do_POST(self):

    form = cgi.FieldStorage(
        fp=self.rfile,
```

```
headers=self.headers,
environ={
    'REQUEST_METHOD': 'POST',
    'CONTENT_TYPE': self.headers['Content-Type'],
})

request = classification_service_pb2.ClassificationRequest()
request.input = form['file'].file.read()

channel = implementations.insecure_channel("127.0.0.1", 9999)
stub = classification_service_pb2.beta_create_ClassificationService_stub(channel)
response = stub.classify(request, 10) # 10 secs timeout

self.respond_form("<div>Response: %s</div>" % response)
```

To run the server we can `python client.py` from outside the container. Then we navigate with a browser to <http://localhost:8080> to access its UI. Go ahead and upload an image to try inference working on it.

Preparing for production

To close the chapter we will learn how to put our classification server in production.

We start by copying the compiled server files to a permanent location inside the container, and cleaning up all the temporary build files:

```
# inside the container
mkdir /opt/classification_server
cd /mnt/home/serving_example
cp -R bazel-bin/. /opt/classification_server
bazel clean
```

Now, outside the container we have to commit its state into a new Docker image. That basically means creating a snapshot of the changes in its virtual file system.

```
# outside the container
docker ps

# grab container id from above
docker commit <container id>
```

That's it. Now we can push the image to our favorite docker serving cloud and start serving it.

Conclusion

In this chapter we learned how to adapt our models for serving, exporting them and building fast lightweight servers that run them. We also learned how to create simple web apps for consuming them giving the full toolset for consuming Tensorflow models from other apps.

In the next chapter we provide code snippets and explanations for some of the helper functions and classes used throughout this book.

Chapter 8. Helper Functions, Code Structure, and Classes

In this short chapter, we provide code snippets and explanations for various helper functions and classes used throughout this book.

Ensure a directory structure

Let's start with a little prerequisite that we need when interacting with the file system. Basically, every time we create files, we have to ensure that the parent directory already exists. Neither our operating system nor Python does this for us automatically, so we use this function that correctly handles the case that some or all of the directories along the path might already exist.

```
import errno
import os

def ensure_directory(directory):
    """
    Create the directories along the provided directory path that do not exist.
    """
    directory = os.path.expanduser(directory)
    try:
        os.makedirs(directory)
    except OSError as e:
        if e.errno != errno.EEXIST:
            raise e
```

Download function

We download several datasets throughout the book. In all cases, there is shared logic and it makes sense to extract that into a function. First, we determine the filename from the URL if not specified. Then, we use the function defined above to ensure that the directory path of the download location exists.

```
import shutil
from urllib.request import urlopen

def download(url, directory, filename=None):
    """
    Download a file and return its filename on the local file system. If the
    file is already there, it will not be downloaded again. The filename is
    derived from the url if not provided. Return the filepath.
    """
    if not filename:
        _, filename = os.path.split(url)
    directory = os.path.expanduser(directory)
    ensure_directory(directory)
    filepath = os.path.join(directory, filename)
    if os.path.isfile(filepath):
        return filepath
    print('Download', filepath)
    with urlopen(url) as response, open(filepath, 'wb') as file_:
        shutil.copyfileobj(response, file_)
    return filepath
```

Before starting the actual download, check if there is already a file with the target name in the download location. If so, skip the download since we do not want to repeat large downloads unnecessarily. Finally, we download the file and return its path. In case you need to repeat a download, just delete the corresponding file on the file system.

Disk caching decorator

In data science and machine learning, we handle large datasets that we don't want to preprocess again every time we make changes to our model. Thus, we want to store intermediate stages of the data processing in a common place on disk. That way, we can check if the file already exists later.

In this section we will introduce a function decorator that takes care of the caching and loading. It uses Python's `pickle` functionality to serialize and deserialize any return values of the decorated function. However, this also means it only works for datasets fitting into main memory. For larger datasets, you probably want to take a look at scientific dataset formats like HDF5.

We can now use this to write the `@disk_cache` decorator. It forwards function arguments to the decorated function. The function arguments are also used to determine whether a cached result exists for this combination of arguments. For this, they get hashed into a single number that we prepend to the filename.

```
import functools
import os
import pickle

def disk_cache(basename, directory, method=False):
    """
    Function decorator for caching pickleable return values on disk. Uses a
    hash computed from the function arguments for invalidation. If 'method',
    skip the first argument, usually being self or cls. The cache filepath is
    'directory/basename-hash.pickle'.
    """
    directory = os.path.expanduser(directory)
    ensure_directory(directory)

    def wrapper(func):
        @functools.wraps(func)
        def wrapped(*args, **kwargs):
            key = (tuple(args), tuple(kwargs.items()))
            # Don't use self or cls for the invalidation hash.
            if method and key:
                key = key[1:]
            filename = '{}-{}.pickle'.format(basename, hash(key))
            filepath = os.path.join(directory, filename)
            if os.path.isfile(filepath):
                with open(filepath, 'rb') as handle:
                    return pickle.load(handle)
            result = func(*args, **kwargs)
            with open(filepath, 'wb') as handle:
                pickle.dump(result, handle)
            return result
        return wrapped
    return wrapper
```

Here is an example usage of the disk cache decorator to save the data processing pipeline.

```
@disk_cache('dataset', '/home/user/dataset/')
def get_dataset(one_hot=True):
    dataset = Dataset('http://example.com/dataset.bz2')
    dataset = Tokenize(dataset)
    if one_hot:
        dataset = OneHotEncoding(dataset)
    return dataset
```

For methods, there is a `method=False` argument that tells the decorator whether to ignore the first argument or not. In methods and class methods, the first argument is the object instance `self` that is different for every program run and thus shouldn't determine if there is a cache available. For static methods and functions outside of classes, this should be `False`.

Attribute Dictionary

This simple class just provides some convenience when working with configuration objects. While you could perfectly well store your configurations in Python dictionaries, it is a bit verbose to access their elements using the `config['key']` syntax.

```
class AttrDict(dict):  
    def __getattr__(self, key):  
        if key not in self:  
            raise AttributeError  
        return self[key]  
  
    def __setattr__(self, key, value):  
        if key not in self:  
            raise AttributeError  
        self[key] = value
```

This class, inheriting from the built-in `dict`, allows to access and change existing elements using the attribute syntax: `config.key` and `config.key = value`. You can create attribute dictionaries by either passing in a standard dictionary, passing in entries keyword arguments, or using `**locals()`.

```
params = AttrDict({  
    'key': value,  
})  
  
params = AttrDict(  
    key=value,  
)  
  
def get_params():  
    key = value  
    return AttrDict(**locals())
```

The `locals()` built-in just returns a mapping from all local variable names in the scope to their values. While some people who are not that familiar with Python might argue that there too much magic going on here, this technique also provides some benefits. Mainly, we can have configuration entries that rely on earlier entries.

```
def get_params():  
    learning_rate = 0.003  
    optimizer = tf.train.AdamOptimizer(learning_rate)  
    return AttrDict(**locals())
```

This function returns an attribute dictionary containing both the `learning_rate` and the `optimizer`. This would not be possible within the declaration of a dictionary. As always, just find a way that works for you (and your colleagues) and use that.

Lazy property decorator

As you learned, our TensorFlow code defines a compute graph rather than performing actual computations. If we want to structure our models in classes, we cannot directly expose its outputs from functions or properties, since this would add new operations to the graph every time. Let's see an example where this becomes a problem:

```
class Model:

    def __init__(self, data, target):
        self.data = data
        self.target = target

    @property
    def prediction(self):
        data_size = int(self.data.get_shape()[1])
        target_size = int(self.target.get_shape()[1])
        weight = tf.Variable(tf.truncated_normal([data_size, target_size]))
        bias = tf.Variable(tf.constant(0.1, shape=[target_size]))
        incoming = tf.matmul(self.data, weight) + bias
        prediction = tf.nn.softmax(incoming)
        return prediction

    @property
    def optimize(self):
        cross_entropy = -tf.reduce_sum(self.target * tf.log(self.prediction))
        optimizer = tf.train.RMSPropOptimizer(0.03)
        optimize = optimizer.minimize(cross_entropy)
        return optimize

    @property
    def error(self):
        mistakes = tf.not_equal(
            tf.argmax(self.target, 1), tf.argmax(self.prediction, 1))
        error = tf.reduce_mean(tf.cast(mistakes, tf.float32))
        return error
```

Using an instance of this from the outside creates a new computation path in the graph when we access `model.optimize`, for example. Moreover, this internally calls `model.prediction` creating new weights and biases. To address this design problem, we introduce the following `@lazy_property` decorator.

```
import functools

def lazy_property(function):
    attribute = '_lazy_' + function.__name__

    @property
    @functools.wraps(function)
    def wrapper(self):
        if not hasattr(self, attribute):
            setattr(self, attribute, function(self))
        return getattr(self, attribute)
    return wrapper
```

The idea is to define a property that is only evaluated once. The result is stored in a member called like the function with some prefix, for example `_lazy_` here. Subsequent calls to the property name then return the existing node of the graph. We can now write the above model like this:

```
class Model:

    def __init__(self, data, target):
        self.data = data
        self.target = target
        self._prediction
```

```

self.optimize
self.error

@lazy_property
def prediction(self):
    data_size = int(self.data.get_shape()[1])
    target_size = int(self.target.get_shape()[1])
    weight = tf.Variable(tf.truncated_normal([data_size, target_size]))
    bias = tf.Variable(tf.constant(0.1, shape=[target_size]))
    incoming = tf.matmul(self.data, weight) + bias
    return tf.nn.softmax(incoming)

@lazy_property
def optimize(self):
    cross_entropy = -tf.reduce_sum(self.target * tf.log(self.prediction))
    optimizer = tf.train.RMSPropOptimizer(0.03)
    return optimizer.minimize(cross_entropy)

@lazy_property
def error(self):
    mistakes = tf.not_equal(
        tf.argmax(self.target, 1), tf.argmax(self.prediction, 1))
    return tf.reduce_mean(tf.cast(mistakes, tf.float32))

```

Lazy properties are a nice tool to structure TensorFlow models and decompose them into classes. It is useful for both node that are needed from the outside and to break up internal parts of the computation.

Overwrite Graph Decorator

This function decorator is very useful when you use TensorFlow in an interactive way, for example a Jupyter notebook. Normally, TensorFlow has a default graph that it uses when you don't explicitly tell it to use something else. However, in a Jupyter notebook the interpreter state is kept between runs of a cell. Thus, the initial default graph is still around.

Excecuting a cell that defines graph operations again will try to add them to the graph they are already in. Fortunately, TensorFlow throws an error in this case. A simple workaround is to *restart the kernel and run all cells again*.

However, there is a better way to do it. Just create a custom graph and set it as default. All operations will be added to that graph and if you run the cell again, a new graph will be created. The old graph is automatically cleaned up since there is no reference to it anymore.

```
def main():
    # Define your placeholders, model, etc.
    data = tf.placeholder(...)
    target = tf.placeholder(...)
    model = Model()

    with tf.Graph().as_default():
        main()
```

Even more conveniently, put the graph creation in in a decorator like this and decorate your main function with it. This main function should define the whole graph, for example defined the placeholders and calling another function to create the model.

```
import functools
import tensorflow as tf

def overwrite_graph(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        with tf.Graph().as_default():
            return function(*args, **kwargs)
    return wrapper
```

This makes the example above a bit easier:

```
@overwrite_graph
def main():
    # Define your placeholders, model, etc.
    data = tf.placeholder(...)
    target = tf.placeholder(...)
    model = Model()

main()
```

This is the end of the chapter, but take a look at the next chapter to read our wrapup of the book.

Chapter 9. Conclusion

You made it! Thank for you reading *TensorFlow for Machine Intelligence*. You should now have a firm understanding of the core mechanics and API for building machine learning models in TensorFlow. If you weren't already knowledgeable about deep learning, we hope that you've gained more insight and comfort with some of the most common architectures in convolutional and recurrent neural networks. You've also seen how simple it can be to put a trained model into a production setting and start adding the power of TensorFlow to your own applications.

TensorFlow has the capabilities to change the way researchers and businesses approach machine learning. With the skills learned in this book, be confident in your ability to build, test, and implement existing models as well as your own newly designed experimental networks. Now that you are comfortable with the essentials, don't be afraid to play around with what's possible in TensorFlow. You now bring a new edge with you in any discussion about creating machine learning solutions.

Next steps and additional resources

Although we've covered much in this book, there are still subjects that simply couldn't fit within these pages. Because of this, we've included a few directions to help get you started diving deeper into TensorFlow.

Read the docs

To a developer who hasn't worked with TensorFlow before, the API documentation may be a little challenging to read due to specific terminology used in TensorFlow. However, now that you're chops are up to snuff, you'll find the API to be invaluable as you craft and code your programs. Keep it open in the background or a separate monitor and you won't regret it:

https://www.tensorflow.org/versions/master/api_docs/index.html

Stay Updated

The absolute best way to keep up-to-date with the latest functionality and features of TensorFlow is the official TensorFlow Git repository on GitHub. By reading pull requests, issues, and release notes, you'll know ahead of time what will be included in upcoming releases, and you'll even get a sense of when new releases are planned.

<https://github.com/tensorflow/tensorflow>

Distributed TensorFlow

Although the basic concepts of running TensorFlow in a distributed setting are relatively simple, the details of setting up a cluster to efficiently train a TensorFlow model could be its own book. The first place you should look to get started with distributed TensorFlow is the official how-to on the tensorflow.org website:

https://www.tensorflow.org/versions/master/how_tos/distributed/index.html

Note that we expect many new features to be released in the near future that will make distributed TensorFlow much simpler and more flexible- especially with regards to using cluster management software, such as Kubernetes.

Building New TensorFlow Functionality

If you want to get under the hood with TensorFlow and start learning how to create your own Operations, we highly recommend the official how-to on tensorflow.org:

https://www.tensorflow.org/versions/master/how_tos/adding_an_op/index.html

The process of building an Operation from scratch is the best way to start getting acquainted with how the TensorFlow framework is designed. Why wait for a new feature when you could build it yourself!

Get involved with the community

The TensorFlow community is active and thriving. Now that you know the software, we highly encourage you to join the conversation and help make the community even better! In addition to the GitHub repository, the official mailing list and Stack Overflow questions provide two additional sources of community engagement.

The TensorFlow mailing list is designed for general discussion related to features, design thoughts, and the future of TensorFlow:

<https://groups.google.com/a/tensorflow.org/d/forum/discuss>

Note that the mailing list is **not** the place to ask for help on your own projects! For specific questions on debugging, best practices, the API, or anything specific, check out Stack Overflow to see if your question has already been answered- if not, ask it yourself!

<http://stackoverflow.com/questions/tagged/tensorflow>

Code from this book

Code examples from the text and additional materials can be found in this book's GitHub Repository:

<https://github.com/backstopmedia/tensorflowbook>

Thank you once again for reading!

Preface

I. Getting started with TensorFlow

1. Introduction

Data is everywhere

Deep learning

TensorFlow: a modern machine learning library

TensorFlow: a technical overview

A brief history of deep learning at Google

What is TensorFlow?

Breaking down the one-sentence description

Beyond the one-sentence description

When to use TensorFlow

TensorFlow's strengths

Challenges when using TensorFlow

Onwards and upwards!

2. TensorFlow Installation

Selecting an installation environment

Jupyter Notebook and Matplotlib

Creating a Virtualenv environment

Simple installation of TensorFlow

Example installation from source: 64-bit Ubuntu Linux with GPU support

Installing dependencies

Installing Bazel

Installing CUDA Software (NVIDIA CUDA GPUs only)

Building and Installing TensorFlow from Source

Installing Jupyter Notebook:

Installing matplotlib

Testing Out TensorFlow, Jupyter Notebook, and matplotlib

Conclusion

II. TensorFlow and Machine Learning fundamentals

3. TensorFlow Fundamentals

Introduction to Computation Graphs

Graph basics

Dependencies

Defining Computation Graphs in TensorFlow

Building your first TensorFlow graph

Thinking with tensors

Tensor shape

TensorFlow operations

TensorFlow graphs

TensorFlow Sessions

Adding Inputs with Placeholder nodes

Variables

Organizing your graph with name scopes

Logging with TensorBoard

Exercise: Putting it together

Building the graph

Running the graph

Conclusion

4. Machine Learning Basics

Supervised learning introduction

Saving training checkpoints

Linear regression

Logistic regression

Softmax classification

Multi-layer neural networks

Gradient descent and backpropagation

Conclusion

III. Implementing Advanced Deep Models in TensorFlow

5. Object Recognition and Classification

Convolutional Neural Networks

Convolution

Input and Kernel

Strides

Padding

Data Format
Kernels in Depth
Common Layers
 Convolution Layers
 Activation Functions
 Pooling Layers
 Normalization
 High Level Layers
Images and TensorFlow
 Loading images
 Image Formats
 Image Manipulation
 Colors
CNN Implementation
 Stanford Dogs Dataset
 Convert Images to TFRecords
 Load Images
 Model
 Training
 Debug the Filters with Tensorboard
Conclusion

6. Recurrent Neural Networks and Natural Language Processing

Introduction to Recurrent Networks
 Approximating Arbitrary Programs
 Backpropagation Through Time
 Encoding and Decoding Sequences
 Implementing Our First Recurrent Network
 Vanishing and Exploding Gradients
 Long-Short Term Memory
 Architecture Variations

Word Vector Embeddings
 Preparing the Wikipedia Corpus
 Model structure
 Noise Contrastive Classifier

Training the model
Sequence Classification
 Imdb Movie Review Dataset
 Using the Word Embeddings
 Sequence Labelling Model
 Softmax from last relevant activation
 Gradient clipping
 Training the model

Sequence Labelling
 Optical Character Recognition Dataset
 Softmax shared between time steps
 Training the Model
 Bidirectional RNNs

Predictive coding
 Character-level language modelling
 ArXiv abstracts API
 Preprocessing the data
 Predictive coding model
 Training the model
 Generating similar sequences

Conclusion

IV. Additional Tips, Techniques, and Features

7. Deploying Models in Production

 Setting up a Tensorflow serving development environment
 Bazel workspace
 Exporting trained models
 Defining a server interface
 Implementing an inference server
 The client app
 Preparing for production
 Conclusion

8. Helper Functions, Code Structure, and Classes

 Ensure a directory structure

- [Download function](#)
- [Disk caching decorator](#)
- [Attribute Dictionary](#)
- [Lazy property decorator](#)
- [Overwrite Graph Decorator](#)

9. Conclusion

[Next steps and additional resources](#)

- [Read the docs](#)
- [Stay Updated](#)
- [Distributed TensorFlow](#)
- [Building New TensorFlow Functionality](#)
- [Get involved with the community](#)
- [Code from this book](#)